

Hardware Description Language Demystified

Explore Digital System Design Using Verilog HDL and VLSI Design Tools

DR. CHERRY BHARGAVA
DR. RAJKUMAR SARMA



VISIT...

LANZAROTE
Caliente.COM

**Hardware
Description
Language Demystified**

*Explore Digital System Design Using
Verilog HDL and VLSI Design Tools*

Dr. Cherry Bhargava

Dr. Rajkumar Sarma



www.bpbonline.com

FIRST EDITION 2020

Copyright © BPB Publications, India

ISBN: 978-93-89898-040

All Rights Reserved. No part of this publication may be reproduced or distributed in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's & publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications

cannot guarantee the accuracy of this information.

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

Ph: 24756967/24756400

BPB BOOK CENTRE

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747

Published by Manish Jain for BPB Publications, 20 Ansari
Road, Darya Ganj, New Delhi-110002 and Printed by him at
Repro India Ltd, Mumbai

www.bpbonline.com

Dedicated to

*My Parents, Husband &
Loving Daughters Mishty & Mauli*

— *Cherry Bhargava*

My Parents and Wife

— *Rajkumar Sarma*

About the Authors

Dr. Cherry Bhargava is working as an associate professor and head, VLSI domain, School of Electrical and Electronics Engineering at Lovely Professional University, Punjab, India. She has more than 14 years of teaching and research experience. She is Ph.D. (ECE), IKGPTU, M.Tech (VLSI Design & CAD) Thapar University and B.Tech (Electronics and Instrumentation) from Kurukshetra University. She is GATE qualified with All India Rank 428.

She has authored about 50 technical research papers in SCI, Scopus indexed quality journals, and national/international conferences. She has eleven books related to reliability, artificial intelligence, and digital electronics to her credit. She has registered five copyrights and filed twenty-two patents. She is the recipient of various national and international awards for being outstanding faculty in engineering and excellent researcher. She is an active reviewer and editorial member of various prominent SCI and Scopus indexed journals. She is a lifetime member of IET, IAENG, NSPE, IAOP, WASET, and reliability research group. Her area of expertise includes the reliability of electronic systems, digital electronics, VLSI design, artificial intelligence, and related technologies.

Dr. Rajkumar Sarma received his B.E. in Electronics and Communications Engineering from Vinayaka Mission's University, Salem, India in 2008. He received his M.Tech as well as PhD degrees from Lovely Professional University, Phagwara, Punjab in the year 2012 & 2020 respectively. He is working as an Assistant Professor in the School of Electronics and Electrical Engineering, Lovely Professional University, Punjab since July 2012. His research interests include Analog and Digital VLSI design, Prototype development using FPGA etc. The author has around 20+ research publications in SCI, Scopus indexed reputed Journals and national or international Conferences. Moreover, the author has 15+ patents published in various engineering fields.

Acknowledgement

At this movement of my substantial enhancement, before we get into the thick of the things, we would like to add a few heartfelt words for the people who gave their constant support with their lousy humor and warm wishes. First and foremost, praises and thanks to God, the Almighty, for His showers of blessings throughout, to complete this book successfully.

We want to acknowledge our students who provided us with the impetus to write a more suitable text. We are thankful to management, seniors, and colleagues of Lovely Professional University for always keep pushing me to move higher and higher. We express my heartfelt gratitude to Dr. Vijay Kumar Banga, Professor, and Principal, Amritsar College of Engineering and Technology, for his consistent guidance, Dr. Shruti Jain, for moral support to accomplish this project. Besides, we thank all our friends, well-wishers, respondents, and academicians, who helped throughout my journey from inception to completion.

Preface

The structure and behavior of electronic circuits are described by HDL. It is a hardware description language that allows analysis and simulation of digital logic and circuits. An IC is created using synthesis, netlist generation, masking, and optimization. The HDL is an integral part of the EDA (electronic design automation) tool for PLDs, microprocessors, and ASICs. The hardware description language was created to implement RTL (register transfer level) abstraction. The widely used HDL is Verilog, and with the advancement of technology, the language is improved at an accelerated rate.

This book is focusing on quickly learning Verilog HDL concepts and, at the same time creating digital system design parallelly. We have shown how to use various VLSI design tools in detail. We start with the basics and get into more complicated stuff with each chapter. Each chapter has exercises that will help you in understanding the concepts clearly.

Over the 14 chapters in this book, you will learn the following:

[Chapter 1: \[An Introduction to VLSI Design Tools\]](#)

This chapter depicts various tools for VLSI design and a stepwise procedure to use different electronic design

automation (EDA) tools, i.e., Xilinx ISE 9.2i, Xilinx Vivado, Cadence NCSIM, etc.

[Chapter 2: \[Need of Hardware Description Language \(HDL\)\]](#)

This chapter explains the need for HDL (Hardware Description Language) and its types. The VLSI design flow and various modeling styles in Verilog HDL are also covered in this chapter

[Chapter 3: \[Logic Gate Implementation in Verilog HDL\]](#)

It explains basic logic gates and their need for digital system design. The implementation of logic gates is also described using Verilog HDL. The concept of logic gate design is further elaborated using enormous programming examples.

[Chapter 4: \[Adder-Subtractor Implementation Using Verilog HDL\]](#)

This chapter explains about the adder, subtractor, adder-cum-subtractor implementation using Verilog HDL. The concept of adder-subtractor design is further elaborated using enormous programming examples.

[Chapter 5: \[Multiplexer/Demultiplexer Implementation in Verilog HDL\]](#)

This chapter explains about the multiplexer and demultiplexer implementation using Verilog HDL. The concept of multiplexer and demultiplexer design is further elaborated using enormous programming examples.

[Chapter 6: \[Encoder/Decoder Implementation Using Verilog HDL\]](#)

This chapter explains about the encoder and decoder implementation using Verilog HDL. The concept of encoder and decoder design is further elaborated using enormous programming examples.

[Chapter 7: \[Magnitude Comparator Implementation Using Verilog HDL\]](#)

This chapter explains the basic of magnitude comparator and its implementation using Verilog HDL. The concept of magnitude comparator and its design are further elaborated using enormous programming examples.

[Chapter 8: \[Flip-Flop Implementation Using Verilog HDL\]](#)

This chapter explains about flip-flop and its types. The conversion of flip-flops is also discussed in this chapter. The implementation of flip-flops using Verilog HDL is also a part of this chapter. The concept of flip-flop design is further elaborated using enormous programming examples.

[Chapter 9: \[Shift Registers Implementation Using Verilog HDL\]](#)

This chapter describes the types, working, and implementation of shift register using Verilog HDL. The concept of SISO,

SIPO, PISO, and PIPO shift register design is further elaborated using enormous programming examples.

[Chapter 10: \[Counter Implementation Using Verilog HDL\]](#)

This chapter describes the types, working, and implementation of various types of counters using Verilog HDL. The concept of up counter, down counter, up/down counter design is further elaborated using enormous programming examples.

[Chapter 11: \[Shift Register Counter Implementation Using Verilog HDL\]](#)

This chapter describes the types, working, and implementation of various types of shift register counters using Verilog HDL. The concept of shift register counters such as Johnson counter and ring counter design is further elaborated using enormous programming examples.

[Chapter 12: \[Advanced Modeling Techniques\]](#)

This chapter describes the advanced modeling techniques in Verilog HDL, such as UDPs (User Defined Primitives). The concept of UDP modeling is further elaborated using enormous programming examples.

[Chapter 13: \[Switch Level Modeling\]](#)

This chapter describes the basics of Complementary Metal Oxide Semiconductor. The implementation of a digital system

is done using switch level modeling in Verilog HDL. The concept of the digital system design is further elaborated using enormous programming examples of switch level modeling.

[Chapter 14: \[FPGA Prototyping in Verilog HDL\]](#)

This chapter describes the basics and needs for reconfigurable computing. The FPGA based prototyping using Verilog HDL is discussed in this chapter. Programming examples of digital design are explained on NEXYS-4 ARTIX-7 FPGA Board.

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors if any, occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at business@bpbonline.com for more details.

At you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

BPB IS SEARCHING FOR AUTHORS LIKE YOU

If you're interested in becoming an author for BPB, please visit www.bpbonline.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at Check them out!

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

IF YOU ARE INTERESTED IN BECOMING AN AUTHOR

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

Table of Contents

1. Introduction to VLSI Design Tools

Structure

Objectives

1.1 Xilinx ISE 9.2i

1.1.1. How to start the tool

1.1.2. Create a new project

1.1.3. Simulation using Xilinx ISE 9.2i

1.2. The VIVADO Design Suite

1.2.1. Creating a new project

1.2.2. Simulation of project

1.3 The Cadence NC-SIM

1.3.1. Creating a new project

1.3.2. Simulating the project

Conclusion

Questions

2. The Need for Hardware Description Language (HDL)

2.1. Introduction to HDL for digital circuits

Structure

Objectives

2.1.1. Need for Hardware Description Language (HDL).

2.2. VLSI Design flow

2.2.1. Digital system design using Verilog HDL

2.3. Modeling styles in Verilog HDL

2.3.1. Gate-level modeling

Verilog HDL using structural modeling/gate-level modeling

Verilog HDL for logic circuits with delays

2.3.2. Data flow modeling

Verilog code using data flow modeling

2.3.3. Behavioral modelling

Verilog code using behavioral modeling

2.3.4. Switch level modeling

Verilog code using switch level modeling

Conclusion

Questions

3. Logic Gate Implementation in Verilog HDL

3.1. Logic gates

Structure

Objectives

3.1.1. AND gate

3.1.1.1 Verilog program for two-input AND gate in dataflow modeling

3.1.1.2 Verilog program for two-input AND gate using structural modeling

3.1.1.3 Verilog program for two-input AND gate in behavioral modeling

3.1.1.4 Verilog program for two input AND gate in switch level modeling

3.1.2. OR gate

[3.1.2.1 Verilog_program_for two-input OR gate in dataflow modeling](#)

[3.1.2.2 Verilog_program_for two-input OR gate in gate-level modeling](#)

[3.1.2.3 Verilog_program_for two input OR gate in behavioral modeling](#)

[3.1.2.4 Verilog_program_for two input OR gate in switch-level modeling](#)

[3.1.3. NOT gate](#)

[3.1.3.1 Verilog_program_for two input NOT gate in dataflow modeling](#)

[3.1.3.2 Verilog_program_for two input NOT gate in gate-level modeling](#)

[3.1.3.3 Verilog_program_for two input OR gate in behavioral modeling](#)

[3.1.3.4 Verilog_program_for two input NOT gate in switch-level modeling](#)

[3.2. Advanced logic gates](#)

[3.2.1. NAND gate](#)

[3.2.1.1 Verilog_program_for two input NAND gate in dataflow modeling](#)

[3.2.1.2 Verilog_program_for two input NAND gate in gate-level modeling](#)

[3.2.1.3 Verilog_program_for two input NAND gate in behavioral modeling](#)

[3.2.1.4 Verilog_program_for two input NAND gate in switch-level modeling](#)

3.2.2 NOR gate

3.2.2.1 Verilog program for two input NOR gate in dataflow modeling

3.2.2.2 Verilog program for two input NOR gate in gate-level modeling

3.2.1.3 Verilog program for two input NAND gate in behavioral modeling

3.2.1.4 Verilog program for two input NOR gate in switch-level modeling

3.2.3. EX-OR or XOR gate

3.2.3.1 Verilog program for two input XOR gate in dataflow modeling

3.2.3.2 Verilog program for two input XOR gate in gate-level modeling

3.2.3.3 Verilog program for two input XOR gate in behavioral modeling

3.2.3.4 Verilog program for two input XOR gate in switch-level modeling

3.2.4. EX-NOR or XNOR gate

3.2.4.1 Verilog program for two input XNOR gate in dataflow modeling

3.2.4.2 Verilog program for two input XNOR gate in gate-level modeling

3.2.4.3 Verilog program for two input XNOR gate in behavioral modeling

3.2.4.4 Verilog program for two input XNOR gate in switch-level modeling

Conclusion

Questions

4. Adder-Subtractor Implementation Using Verilog HDL

4.1. Introduction

Structure

Objectives

4.2. The design strategy of combinational logic circuits

4.3. Adders

4.3.1. Half Adder

4.3.2. Full adder

4.3.3. Verilog program for half adder in dataflow modelling

4.3.4. Verilog program for full adder using half adder

4.4. Subtractor

4.4.1. Half subtractor

4.4.2. Full subtractor

4.4.3. Verilog program for half subtractor in behavioral modelling

4.4.4. Verilog program for full subtractor using half subtractor

4.4.5. Verilog program for full subtractor using half subtractor

4.5. Verilog code for adder cum subtractor

Conclusion

Questions

5. Multiplexer/Demultiplexer Implementation in Verilog HDL

5.1 Introduction to multiplexer

Structure

Objectives

5.2 The 4:1 Multiplexer

5.3. IC 74153 pin diagram

5.4. Verilog program for 2:1 mux design

5.4.1. Multiplexer implementation using dataflow modeling

5.4.2. Multiplexer implementation using gate-level modelling

5.4.3. Multiplexer implementation using behavioral modeling

5.5. Verilog program for 4:1 mux design

5.5.1. Multiplexer implementation using dataflow modelling

5.5.2. Multiplexer implementation using structural modelling

5.6. Verilog program for 8:1 mux design

5.6.1. Multiplexer implementation using structural modelling

5.7. Introduction to demultiplexer

5.7. The 1:2 demultiplexer

5.8. The 1:4 demultiplexer

5.9. Verilog program for 1:2 demux design

5.9.1. Demultiplexer implementation using dataflow modelling

5.9.2. Demultiplexer implementation using gate-level modelling

5.9.3. Demultiplexer implementation using behavioral modelling

5.10. Verilog program for 1:4 demux design

5.10.1. Demultiplexer implementation using dataflow modelling

5.10.2. Demultiplexer implementation using gate-level modelling

5.11. Verilog program for 1:8 demux design

5.11.1. Demultiplexer implementation using behavioral modeling

Conclusion

Questions

6. Encoder/Decoder Implementation Using Verilog HDL

6.1. Introduction to encoder

Structure

Objectives

6.2. Octal to binary encoder

6.3. Decimal to BCD encoder

6.4. Verilog program for 4:2 encoder

6.4.1. The 4:2 encoder design using dataflow modeling

6.4.2. The 4:2 encoder design using behavioral modeling

6.5. Priority encoder

6.6. Verilog code for priority encoder design

6.7. Introduction to decoder

6.8. The 2 line to 4 line decoder

6.9. The 3 line to 8 line decoder

6.10. Verilog program for 2:4 decoder design

6.10.1. Decoder implementation using dataflow modelling

6.10.2. Decoder implementation using gate-level modelling

6.10.3. Decoder implementation using behavioral modelling

6.11. Verilog program for 3:8 decoder design

6.11.1. Decoder implementation using behavioral modelling

6.11.2. The 3:8 decoder implementation using 2:4 decoder

Conclusion

Questions

7. Magnitude Comparator Implementation Using Verilog HDL

7.1. Introduction

Structure

Objectives

7.2. One-bit magnitude comparator

7.2.1. One bit magnitude comparator using logic gates

7.3. Two-bit magnitude comparator

7.3.1. Two-bit magnitude comparator using logic gates

7.3.2. N-bit magnitude comparator using Verilog HDL

Conclusion

Questions

8. Flip-Flop Implementation Using Verilog HDL

8.1. Introduction to flip-flop

Structure

Objectives

8.1.1. Types of flip-flop

8.2. SR flip-flop

8.2.1. Characteristics table of SR flip-flop

8.2.2. Excitation table of SR flip-flop

8.3. JK flip-flop

8.3.1. Characteristics table of JK flip-flop

8.3.2. Excitation table of JK flip-flop

8.4. Master-slave JK flip-flop

8.4.1. Working of a master-slave flip-flop

8.5. D flip-flop

8.5.1. Characteristics table of D flip-flop

8.5.2. Excitation table of D flip-flop

8.6. T flip-flop

8.6.1. Characteristics table of T flip-flop

8.6.2. Excitation table of T flip-flop

Conclusion

Questions

9. Shift Registers Implementation Using Verilog HDL

9.1. Introduction to shift registers

Structure

Objectives

9.1.1 Classification of shift registers

9.2. Serial-In-Serial-Out (SISO) shift register

9.3. Serial-In-Parallel-Out (SIPO) shift register

9.4. Parallel-In-Serial-Out (PISO) shift register

9.5. Parallel-In-Parallel-Out (PIPO) shift register

Conclusion

Questions

10. Counter Implementation Using Verilog HDL

10.1. Introduction to Counters

Structure

Objectives

10.1.1. Classification of counters

10.2. Asynchronous counters

10.2.1. MOD-4/2-bit asynchronous counter/ripple counter

10.2.2. Advantages & disadvantages of asynchronous counters

10.2.3. Verilog program of the 2-bit asynchronous counter using the positive edge-triggered T flip-flop

10.3. Synchronous counter

10.3.1. Mod-4/2-bit synchronous counter using JK flip-flop

10.3.2. 3-bit up/down counter using T flip-flop

10.3.3. Verilog program of 2-bit synchronous counter using positive edge triggered D flip-flop

10.3.4. Verilog program of 4-bit up/down counter in behavioural modelling

Conclusion

Questions

11. Shift Register Counter Implementation Using Verilog HDL

11.1. Introduction to shift register counters

Structure

Objectives

11.1.1 Types of shift register counters

11.1.1.1 Ring counter

11.1.1.2 Johnson counter

11.2. Count the number of ones in a bit sequence using

Verilog HDL

11.2.1. Implementation using TASK

11.2.2. Implementation using FUNCTION

Conclusion

Questions

12. Advanced Modelling Techniques

12.1. Introduction to UDP

Structure

Objectives

12.1.1. Types of UDP

12.2. Syntax of UDP

12.3. Rules of UDP

12.4. Verilog HDL program using UDP

12.4.1. Verilog code for 4x1 multiplexer using UDP

12.4.2. Verilog code for JK flip-flop using UDP

Conclusion

Questions

13. Switch Level Modelling

13.1. Introduction to switch level modeling

Structure

Objectives

13.2. Switch level primitives

13.2.1. MOS switches

13.2.2. CMOS switches

13.3. Switch level modelling using Verilog HDL

13.3.1. Two-input NAND gate using PMOS/NMOS switches

13.3.2. Verilog HDL program in switch level modelling for implementing the logical expression, $Y = AB + C$

13.3.3. Verilog HDL program in switch level modelling (using TG) for implementing the logical expression, $Y = AB + C$

Conclusion

Questions

14. FPGA Prototyping in Verilog HDL

14.1. Introduction to HDL programming

Structure

Objectives

14.2. Programming FPGA board

14.3. Example of digital design on Nexys-4 Artix-7 FPGA board

14.3.1. Half adder

14.3.2. HDL source file

14.3.3. HDL top module

14.3.4. XDC file

Conclusion

Questions

CHAPTER 1

Introduction to VLSI Design Tools

As technology advances, the number of components per unit chip is also increasing at the accelerated speed, which makes the designing and placing the components a challenging issue. For reducing human effort and enhancing accuracy, various automated tools are used in the VLSI design industry. This chapter depicts a stepwise procedure to use different electronic design automation (EDA) tools, i.e., Xilinx ISE 9.2i, Xilinx Vivado, Cadence NCSIM, etc.

Structure

In this chapter, we will discuss the following topics:

Introduction to VLSI design tools.

Step-by-step procedure to create a project on Xilinx
ISE/VIVADO/Cadence NC-Sim

The process to simulate the HDL program on Xilinx
ISE/VIVADO/Cadence NC-Sim

Objectives

After studying this unit, you should be able to:

Learn various **Computer-Aided Design (CAD)** tools to design digital systems using HDL

Simulate & synthesize an HDL project using Xilinx ISE/VIVADO/Cadence NC-Sim

1.1 Xilinx ISE 9.2i

The Xilinx 9.2i version includes all the features and enhancements of Xilinx ISE Foundation 9.2i software with full support for optional embedded, **digital signal processing (DSP)**, and real-time debug design flows. Here ISE stands for Integrated Software Environment. ISE WebPACK 9.2i software expands development options for FPGA designers with support for Microsoft Windows Vista. ISE WebPACK 9.2i is the only complete FPGA design suite offering a free, downloadable solution, with RTL simulation, for Microsoft Windows Vista as well as Microsoft Windows XP and Linux. This version of Xilinx includes enhanced support for the company's leading 65nm Virtex-5 FPGAs with the addition of three new devices; the Virtex-5 LX30T, LX50, and LX50T. The 9.2i release also includes support for the company's latest high volume Spartan-3 generation FPGAs including the DSP optimized Spartan-3A DSP platform FPGA. In total, ISE WebPACK 9.2i provides support for ten additional Xilinx FPGAs.

1.1.1. How to start the tool

To start the Xilinx ISE 9.2i, click on **Start** → **All programs** → **Xilinx ISE 9.2i** → **Project** as shown in [figure](#)

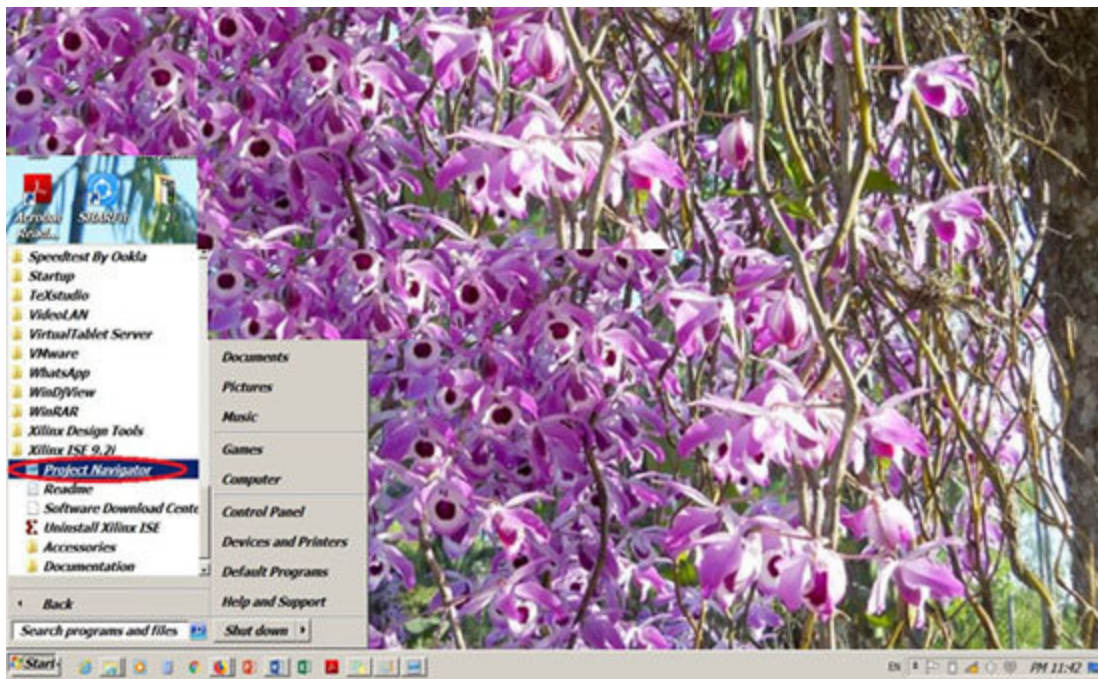


Figure 1.1: Link to open the Xilinx ISE 9.2i

The Xilinx ISE 9.2i landing page consists of four significant windows. The windows are **Process Window**, **Sources Window**, **Console Window & Editor Window**.

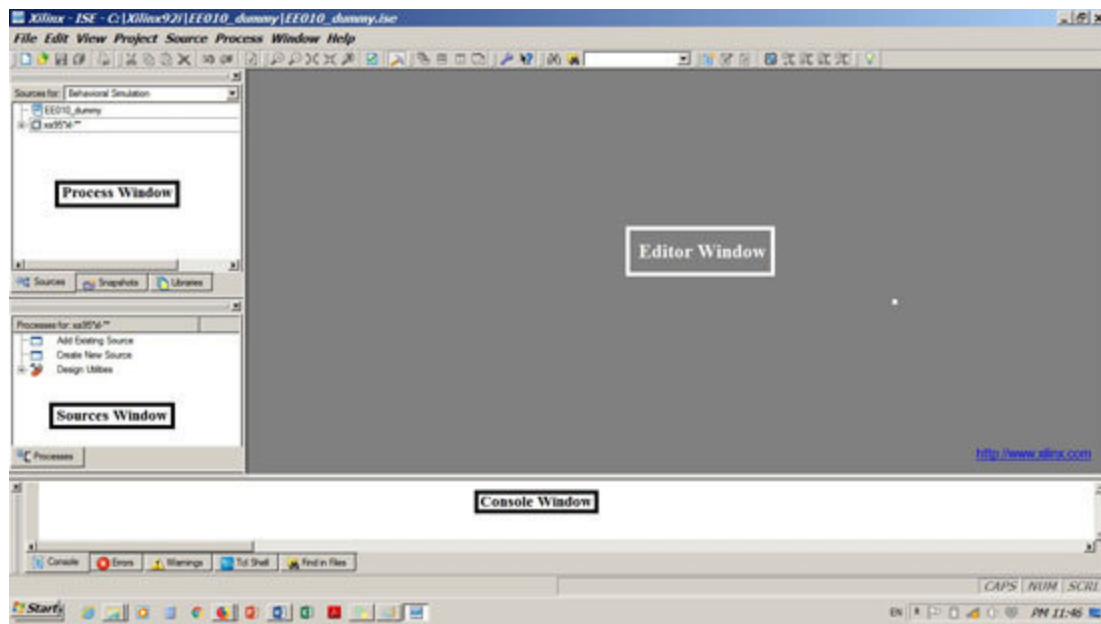


Figure 1.2: Different windows in Xilinx ISE 9.2i

Process It shows the project details such as project name, associated device name, top module or test bench, main module & it's submodules (as instances).

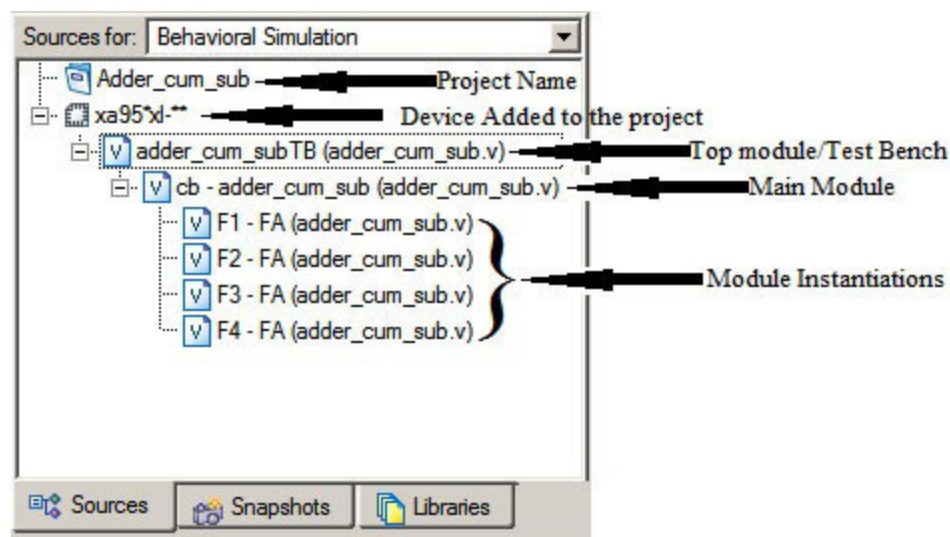


Figure 1.3: *Process window view with sources in a hierarchy*

Moreover, process windows also offer to choose different simulations such as Synthesis/Implementation, Behavioral Simulation & Post-fit Simulation.

Synthesis/Implementation is used to run the design synthesis. It converts the behavioral description to its equivalent gate-level netlist.

Behavioral Simulation: For simulating the design and verifying the output waveform, Behavioral Simulation is performed.

Post-fit The main objective for running the Post-fit Simulation is to verify the design functionality.

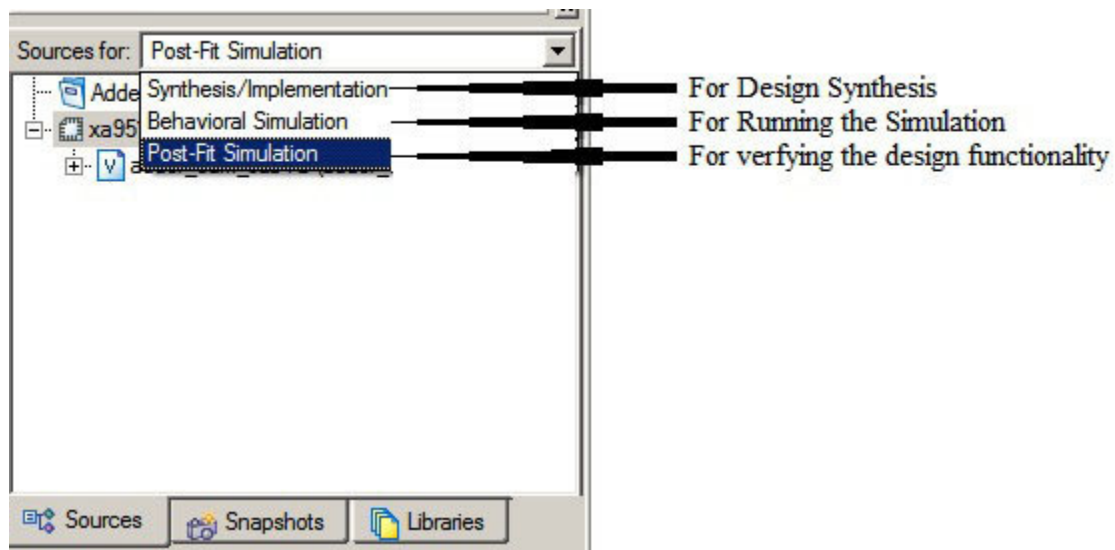


Figure 1.4: Process window for different simulations

To start with a new project, click on **File** → **New Project** & write the name of the project, as shown in [figures 1.5](#) & Here the selection of proper directory & the top-level source type is essential. The top-level source type must be HDL if one is using Verilog or VHDL for the designs. Click on **Next** to continue.

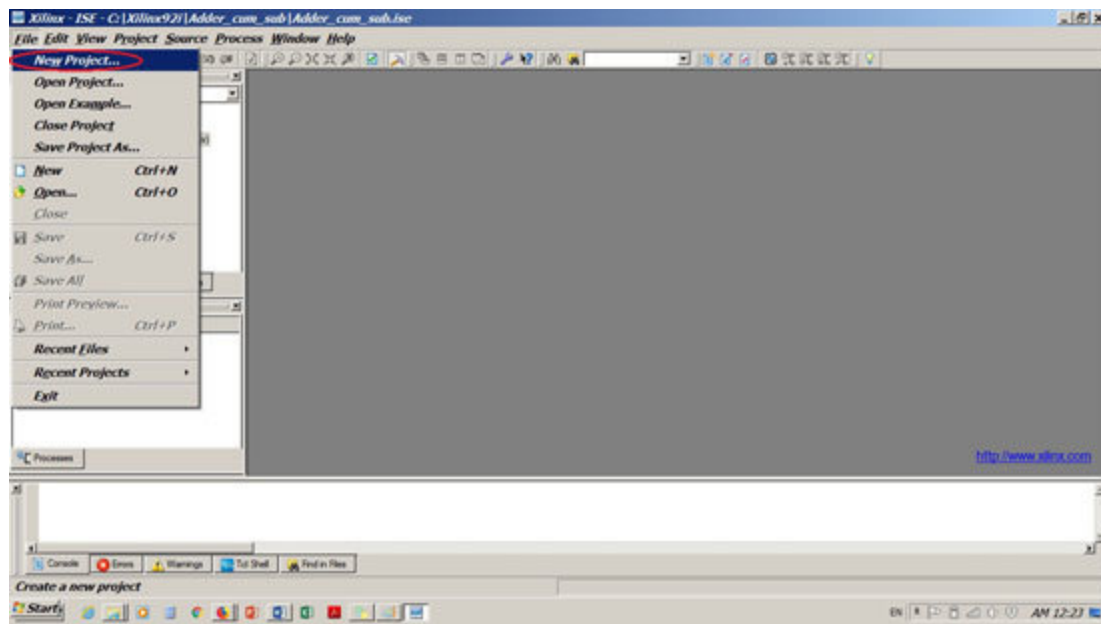


Figure 1.5: Creation of a new project

Choose the correct product category, family & device. For running the simulation, the product category, family & device are not that important & therefore keep it as the default value only. The only vital field to choose in this window is the **Preferred** The programmer may choose **Verilog** or **VHDL** from the drop-down, as shown in [figure](#) To continue, click on the **Next** button.

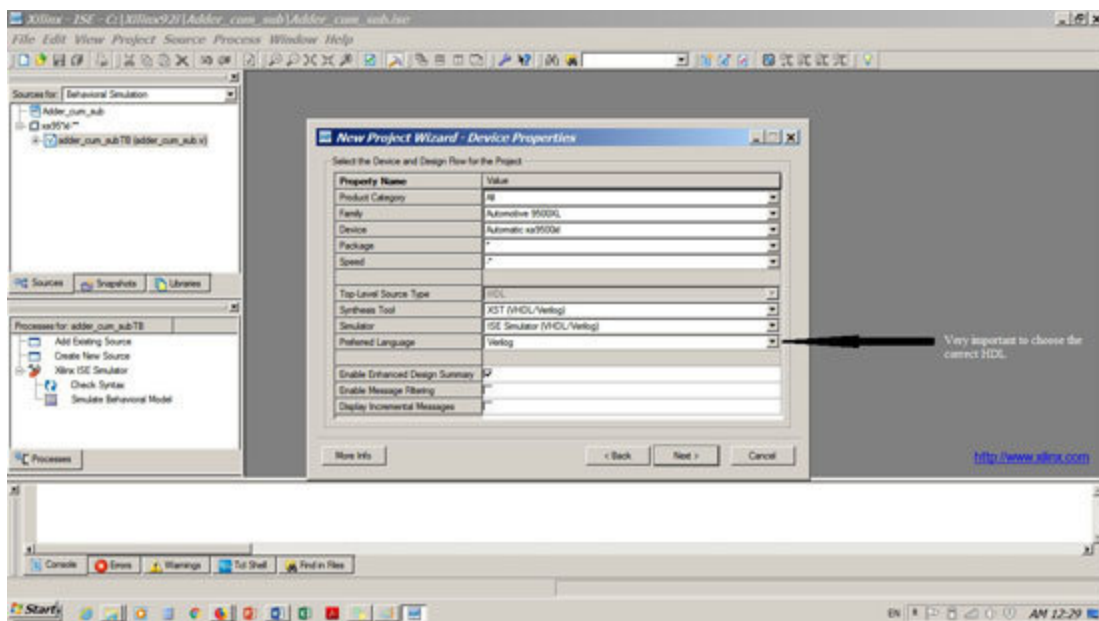


Figure 1.7: Selection of proper language, Logic simulator, logic family, etc.

To add a new source, click on the **New Source** button & click on The **New Source Wizard** shows a range of source types such as the **Verilog module**, **Verilog Test Fixture**, **VHDL module**, **VHDL test** etc. as shown in [figure](#)

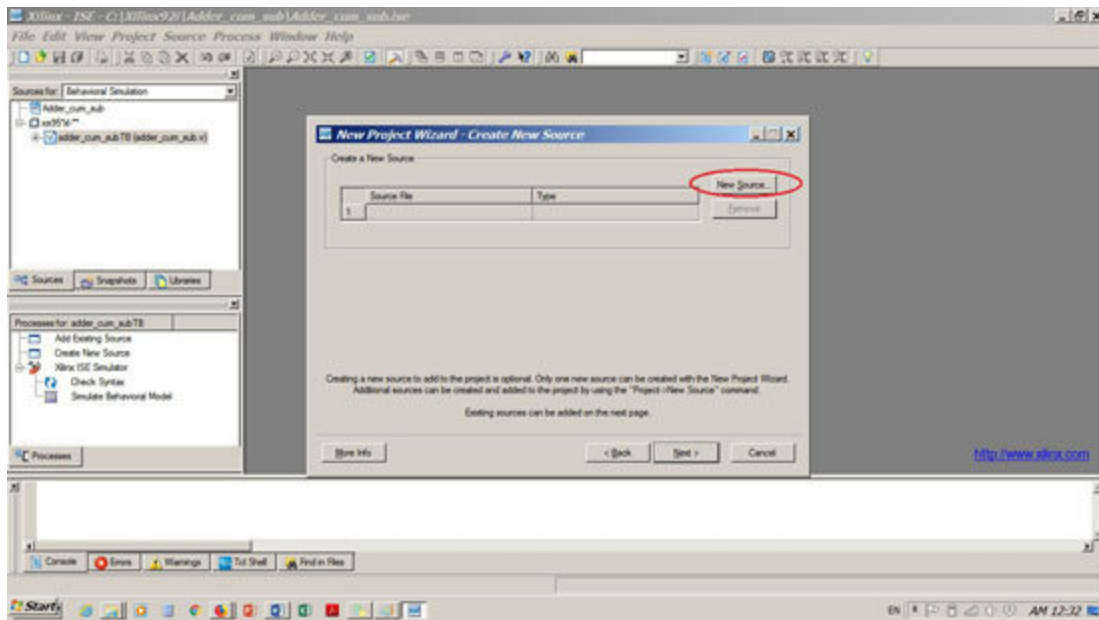


Figure 1.8: To add a new source

For simplicity, choose the **Verilog module** or **VHDL module** (based on your HDL preference), type the name of the file & click on The wizard also shows the location of the file on the hard disk. It is essential to note that there is no need to provide an extension to the HDL files. The extensions of the files (.v for Verilog & .vhd for VHDL) will be automatically added by the wizard once the file is created.

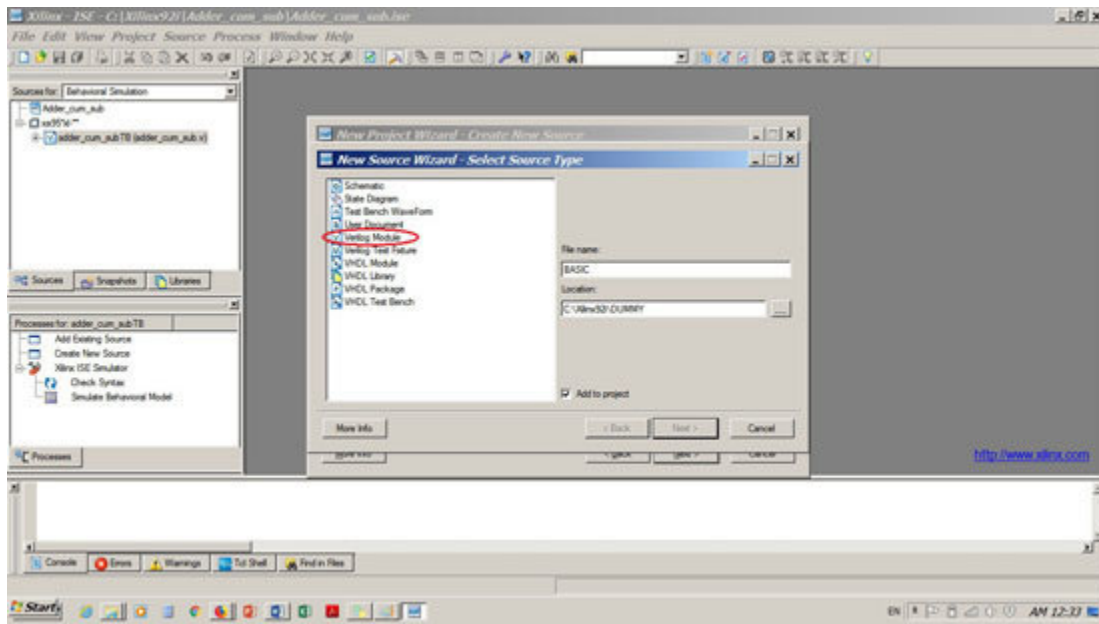


Figure 1.9: To create a new module

In the next window, type the port name, direction & its bit size (if any). The size of the port (bits) can be specified in the **LSB** & **MSB** tabs and click on

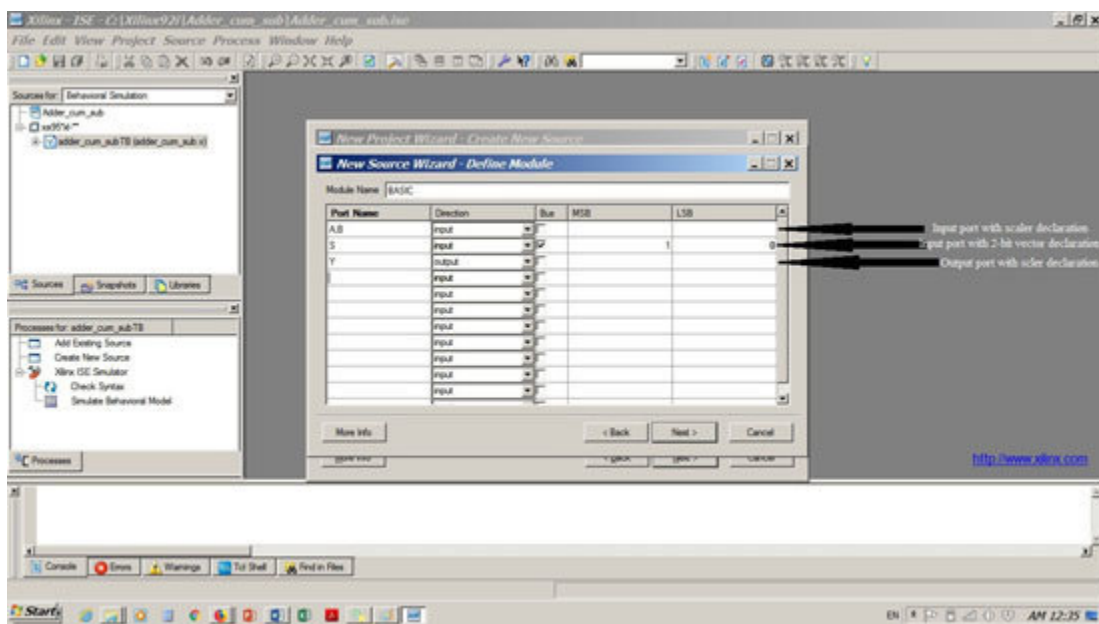


Figure 1.10: To provide the port list & directions

The final summary window will show the summary of the news source added, as shown in [figure](#). Verify the same and click on the

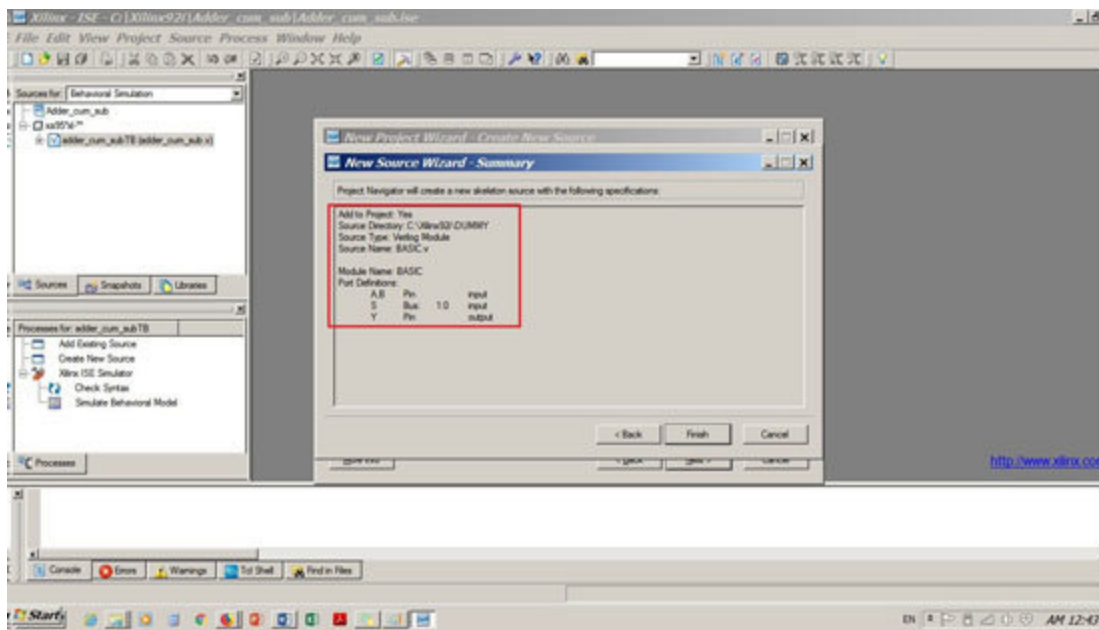


Figure 1.11: Summary of the source added

As the file is created for the first time, the wizard will ask to create a new directory. Hit **Yes** & click on

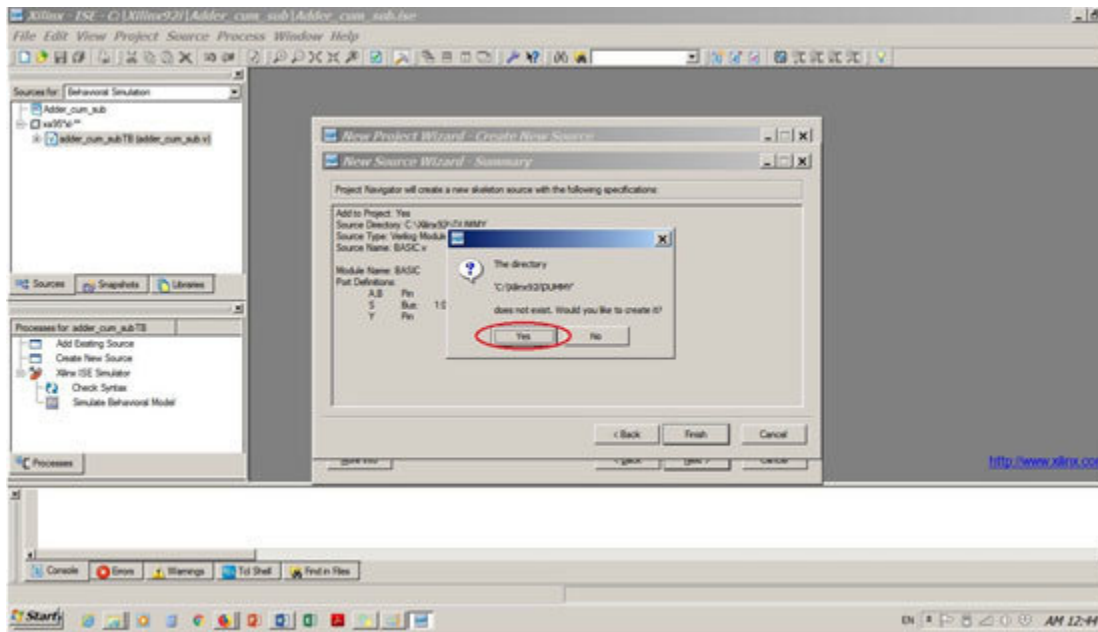


Figure 1.12: Confirmation to create a new directory

Once the newly added source is defined along with its input, output & input ports, the wizard asks for further confirmation from the user to proceed further. The user may continue using the newly added source or/and he may add an existing source file (HDL) already available in a different project. For browsing the existing file, just click on the **Browse** button & select the desired source file, as shown in [figure](#)

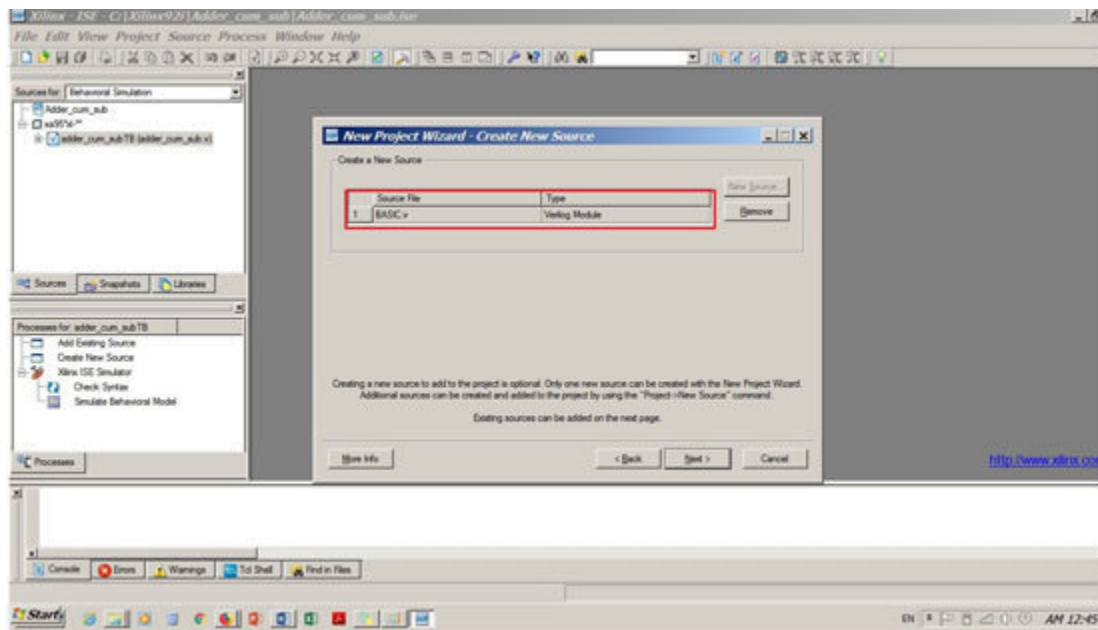


Figure 1.13: New project wizard with newly added source

To add an already created source, click on the **Add source** tab & the browser will open up.

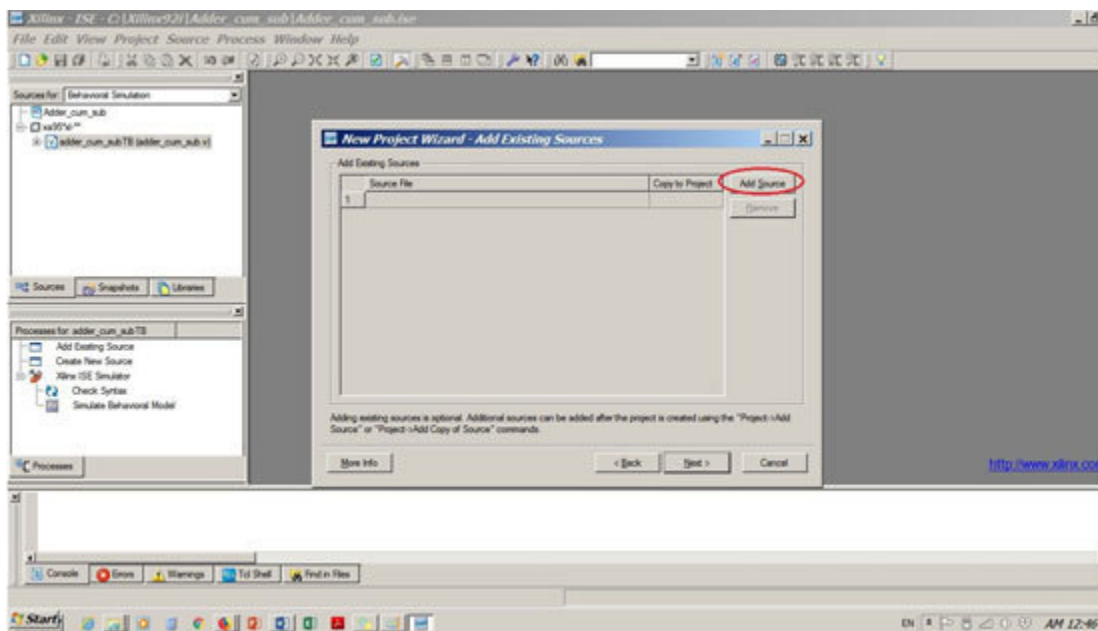


Figure 1.14: To add an existing HDL

Choose the desired V/VHL file & click on **Once** the file is added to the project, click on the **Next** button. The snapshots of the same are shown in [figures 1.14_&](#)

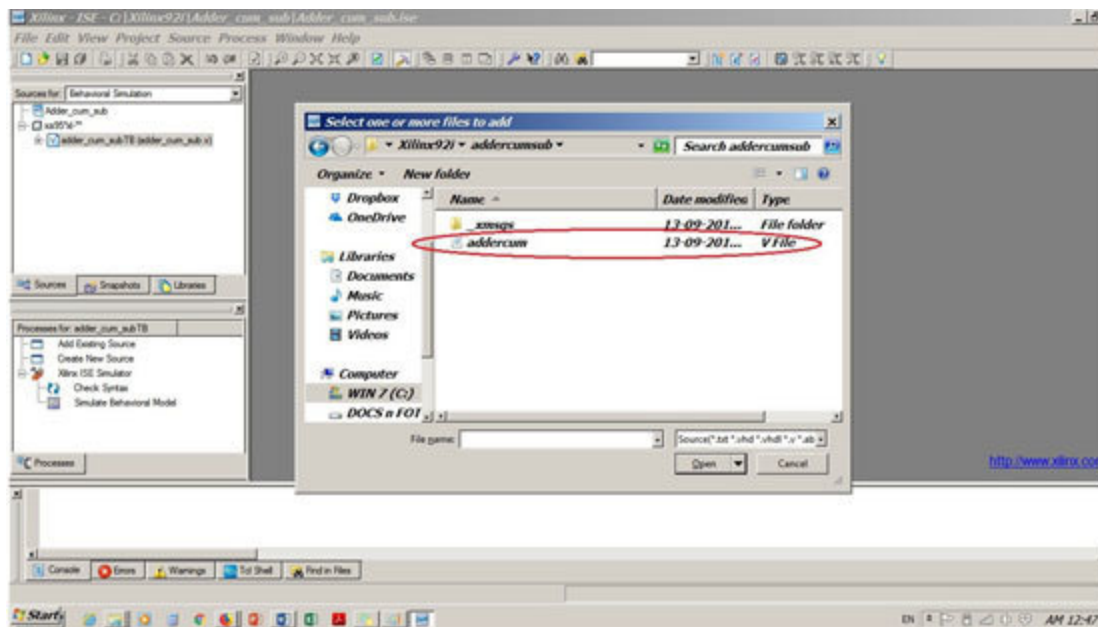


Figure 1.15: To browse an existing HDL file

Once completed adding a new source file or an existing source file, click on **Finish** to create the project.

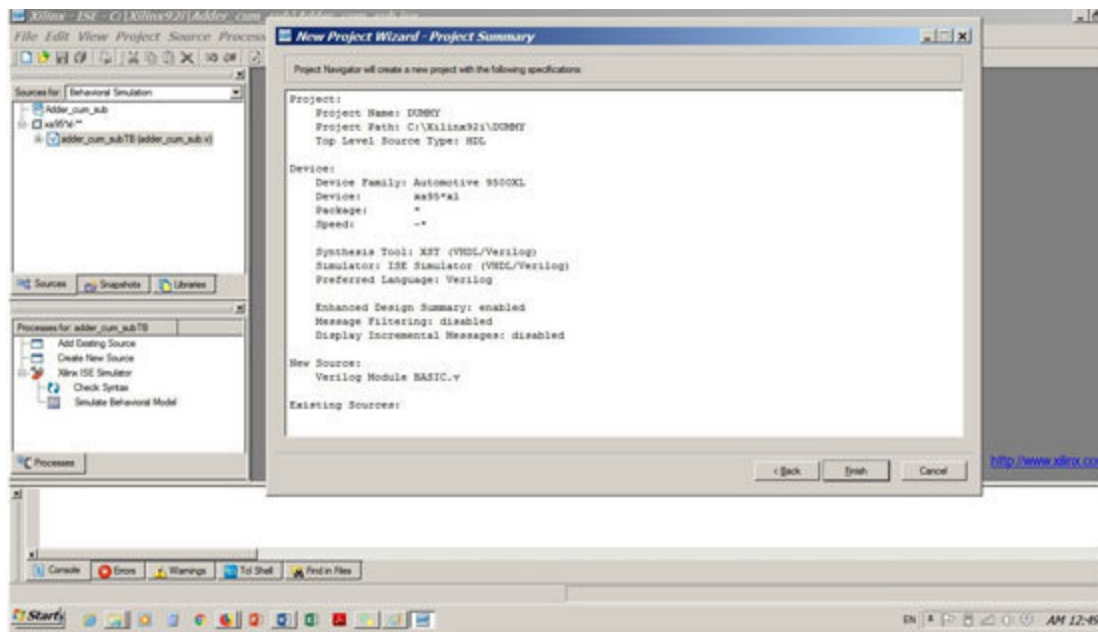


Figure 1.16: Summary of the new project

After clicking on the **Finish** button, the editor window will open up with the skeleton of the source file that is created. The source file will be opened in the form of a module (in Verilog) or entity (in VHDL) having the direction of the ports but without any internal architecture, as shown in [figure](#)

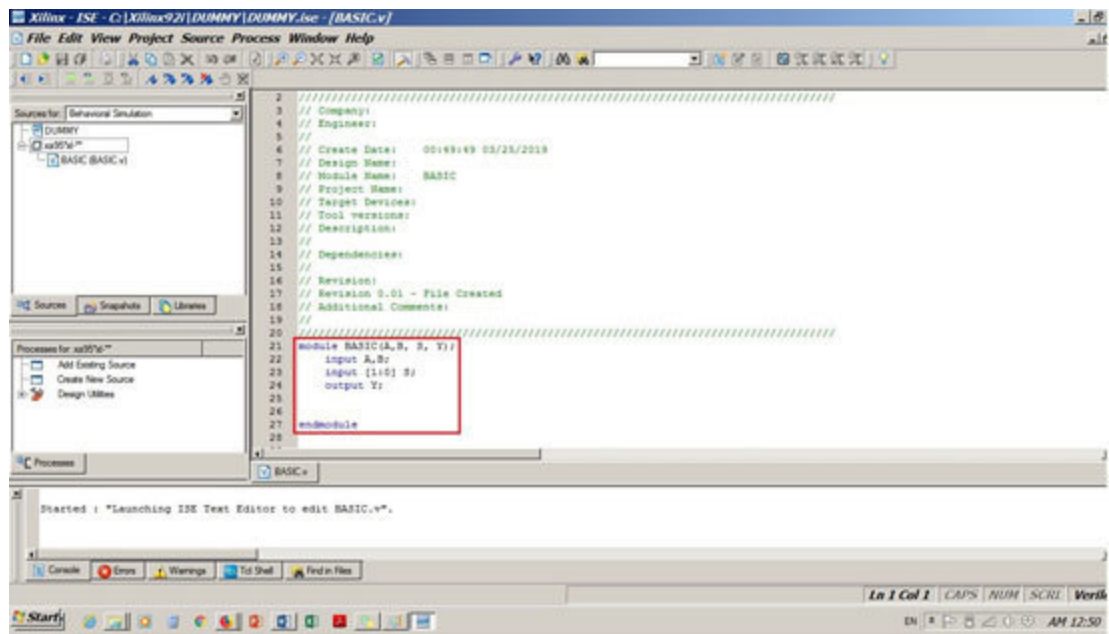


Figure 1.17: To edit the source file

1.1.3. Simulation using Xilinx ISE 9.2i

Immediately after creating the project, a new text editor window will open up for editing the source file. The critical thing to notice here is that the port list mentioned in the **Add a new source** wizard will be automatically added to the module & the module name will be the same as the file name. It is not at all mandatory to keep the module name or ports as it is mentioned in the wizard. If someone wishes to change the module name or list of the port, he/she may do so.

In the same V/VHL files, multiple modules can be created, such as the main module & the test bench modules.

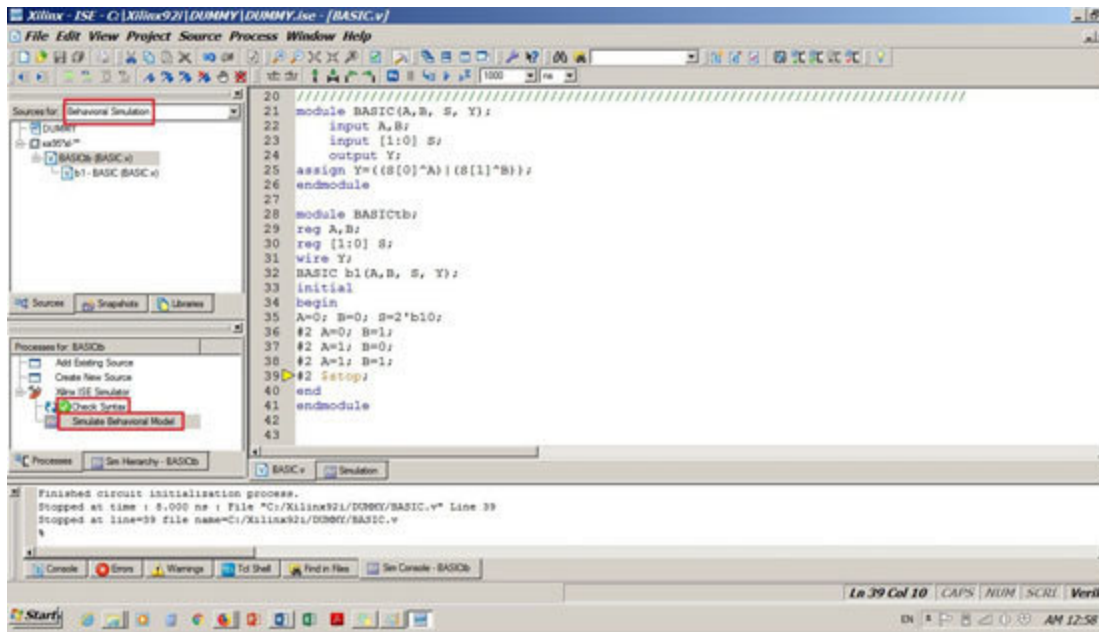


Figure 1.18: To add a new module in the existing source file

Once the modules are designed, the next step is to run the simulation. For running the simulation, choose **Behavioral simulation** in the **Sources for** window & select the test bench module name. Then go to the process window & open the Xilinx ISE simulator. To validate the code, do the following:

Click on **Check syntax** to check error in the code

Click on **Simulate behavioral model** for running the simulation

Once the simulation is run, the simulation waveform will be viewed in a different tab. The **Hierarchy** tab will show different

signals/registers of the test bench as well as the main module.

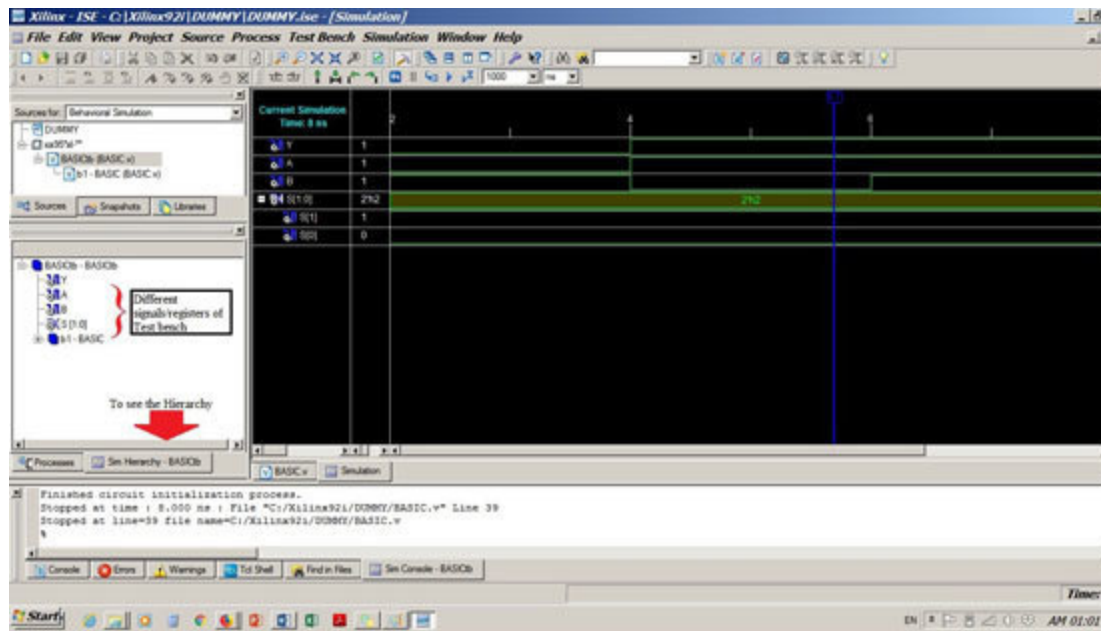


Figure 1.19: *Simulating procedure*

For verifying the simulation at an instance, as shown in [Figures 1.19](#) and use the marker to view the input as well as output values.

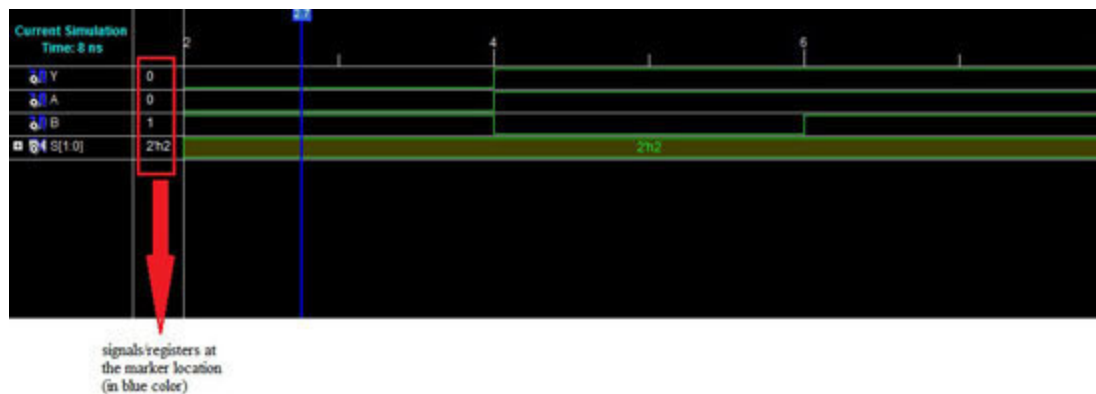


Figure 1.20: How to read the waveform

Moreover, to view the content of a vector input/output, click on the + symbol to view the content of the vector bitwise, as shown in [figure](#)

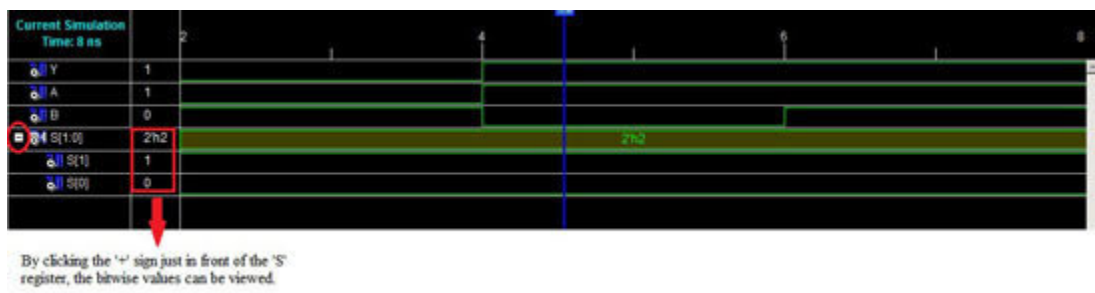


Figure 1.21: How to read the waveform in vector form

1.2. The VIVADO Design Suite

The VIVADO Design Suite is a software suite produced by Xilinx for synthesis and analysis of HDL designs, superseding Xilinx ISE with additional features for system on chip development and high-level synthesis. Vivado represents a ground-up rewrite and re-thinking of the entire design flow (compared to ISE) and has been described by reviewers as well-conceived, tightly integrated, blazing-fast, scalable, maintainable, and intuitive. Unlike ISE, which relied on ModelSim for simulation, the Vivado System Edition includes an in-built logic simulator. Vivado also introduces high-level synthesis, with a toolchain that converts C code into programmable logic. Vivado has been described as a “state-of-the-art comprehensive EDA tool with all the latest bells and whistles in terms of the data model, integration, algorithms, and performance.”

Vivado is an EDA tool developed by Xilinx, which targets the prototype development on FPGA boards. The step by step process to simulate on Vivado 2017.1 is as follows:

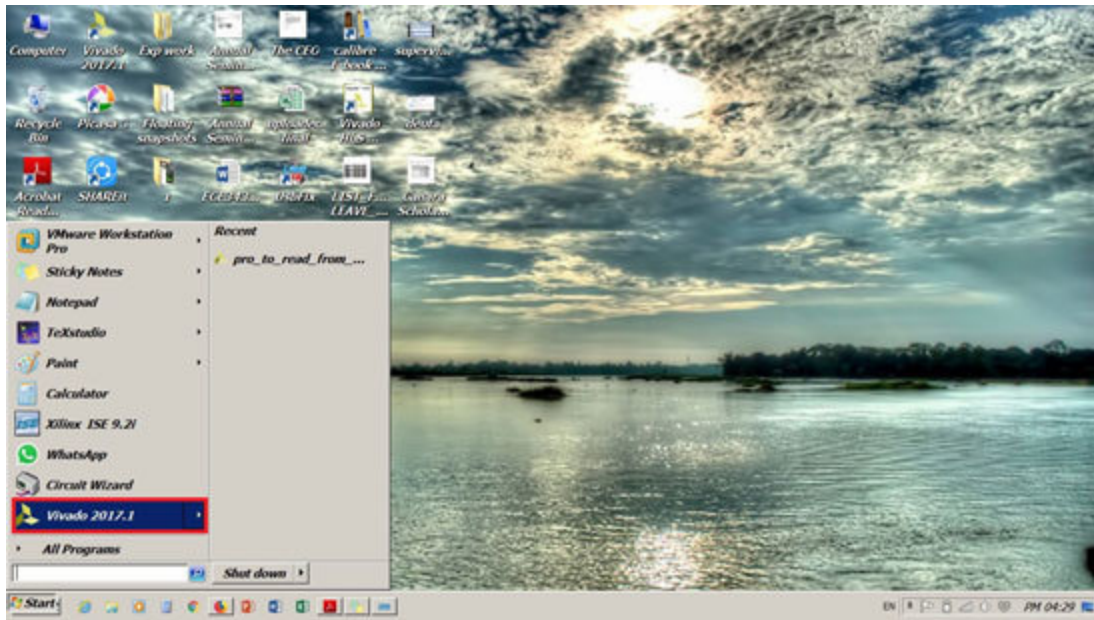


Figure 1.22: Link to open the Xilinx Vivado

Open the tool by clicking on the **Vivado 2017.1** icon available at the Desktop or the startup menu.



Figure 1.23: Welcome message while opening Xilinx Vivado

To begin with, click on the **Create Project** option available at the landing page of the tool, as shown in [figure](#). The projects that are done earlier may be accessed from the **Open Project** option. Moreover, Xilinx has provided some of the example projects as well. For accessing the same, one can click on the **Open Example Project** option.

1.2.1. Creating a new project

To start working on VIVADO, creating a new project is the first step. For doing the same, click on **Create Project** on the landing page of the tool or go to **File** → **Create** as shown in [figure](#)

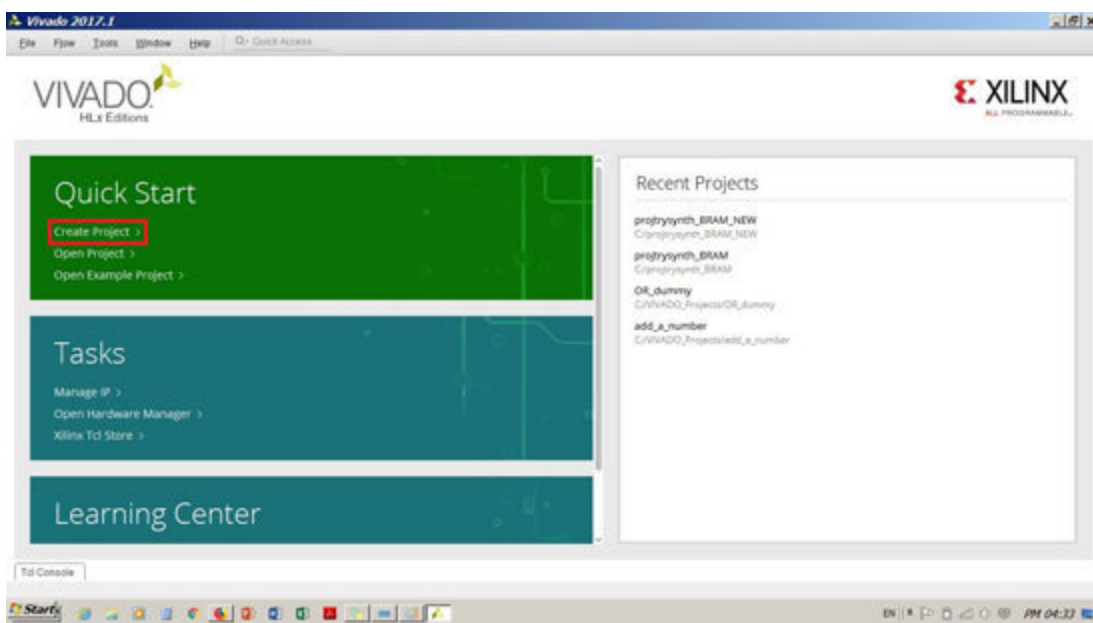


Figure 1.24: Link to create a new project

Click on the **Next** button to continue.

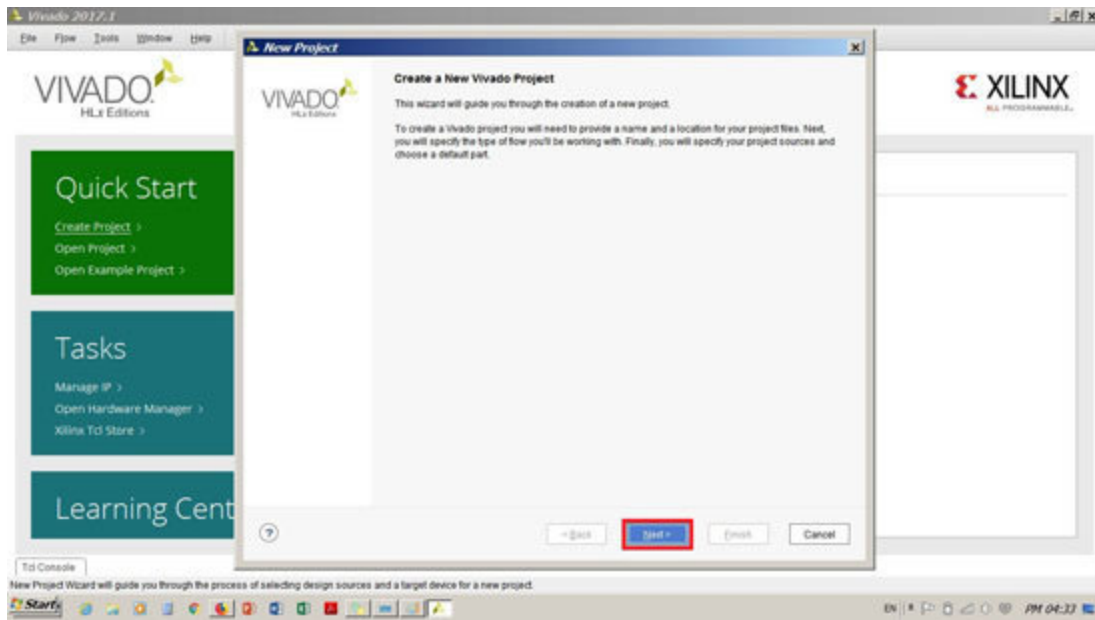


Figure 1.25: *New project wizard*

Provide the project name & the project directory & click on as shown in [figure](#)

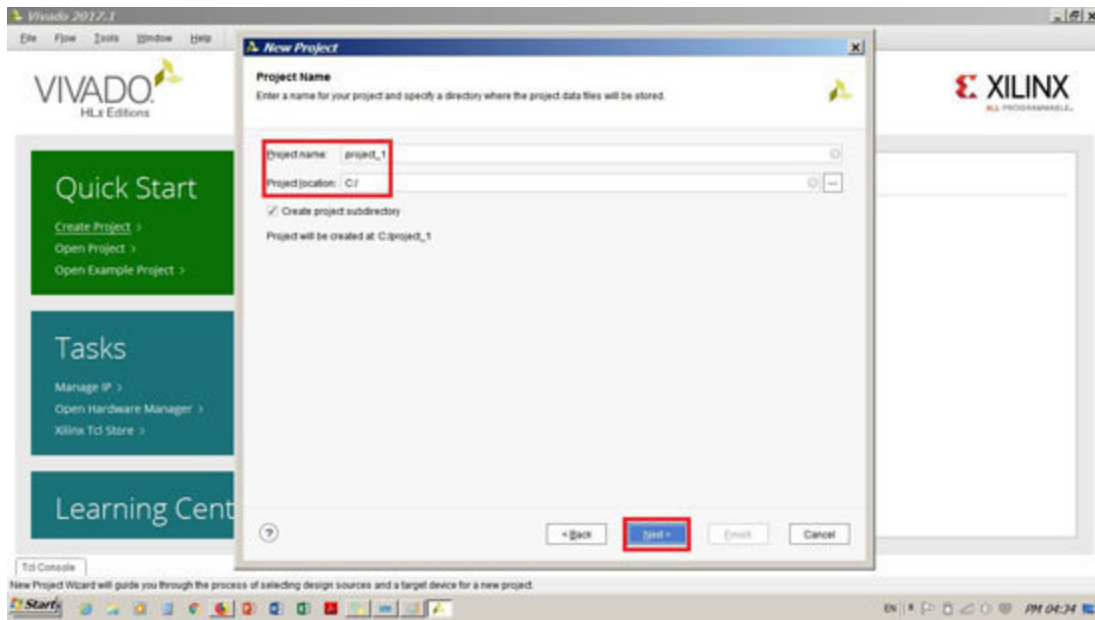


Figure 1.26: adding a new project

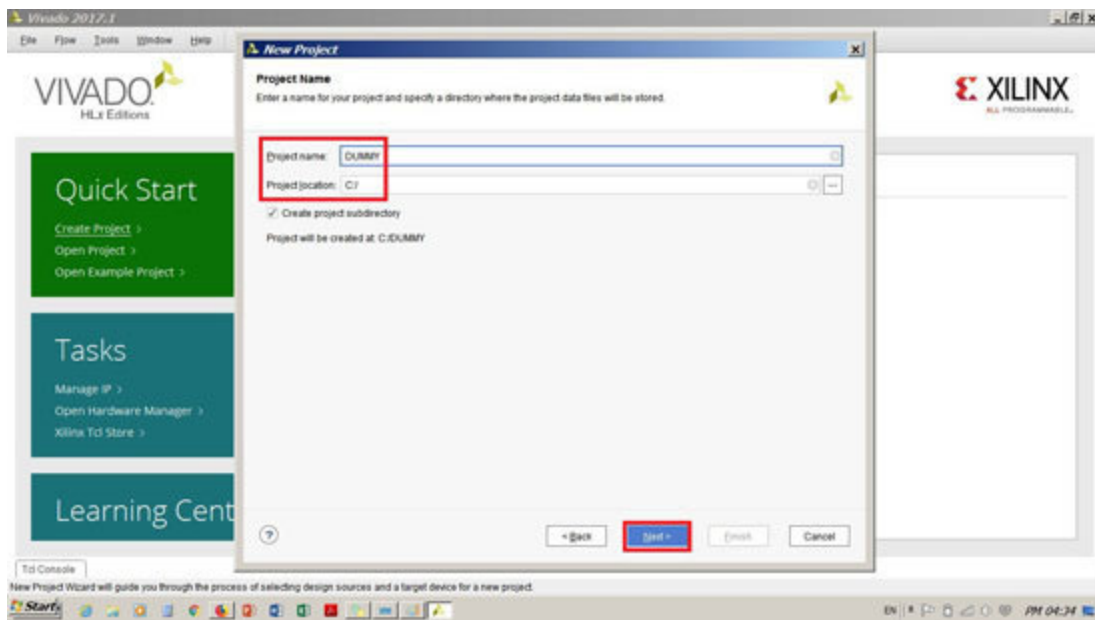


Figure 1.27: Typing the project name

To create a new source file, click on the **Create File** button. One thing to remember here is that one source file may consist of multiple modules (including the test bench). Therefore, there is no requirement to create multiple source files for multiple modules.

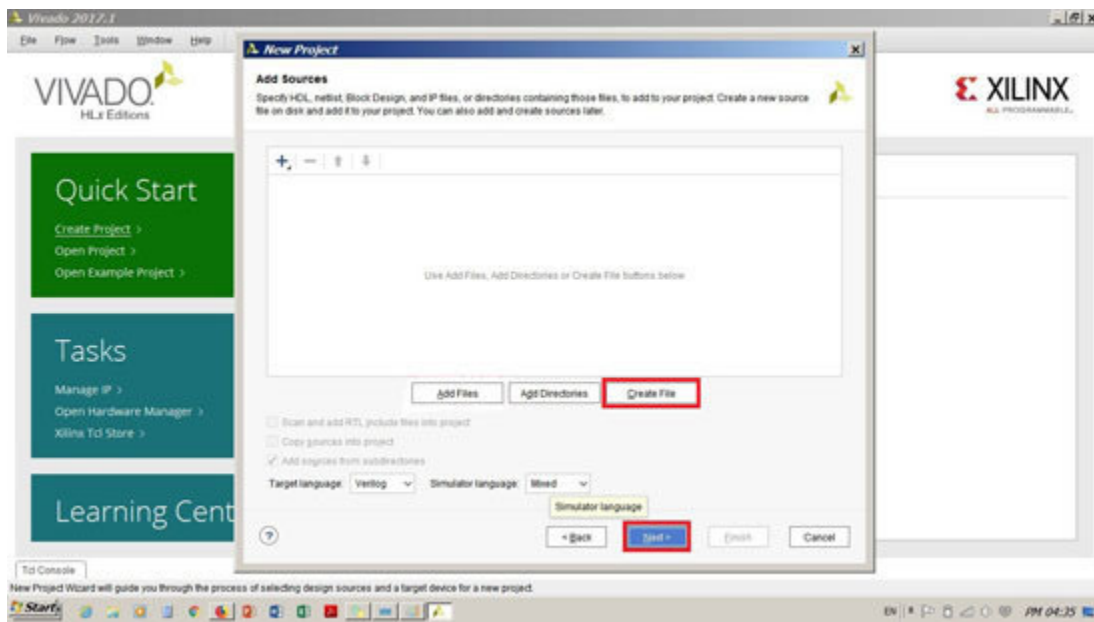


Figure 1.28: To add a new source

A **Create Source File** wizard will open, as shown in [figure](#). Provide the file name and choose the appropriate file type. Choosing the appropriate file type is very important because, in the drop-down list, there are plenty of other options as well.

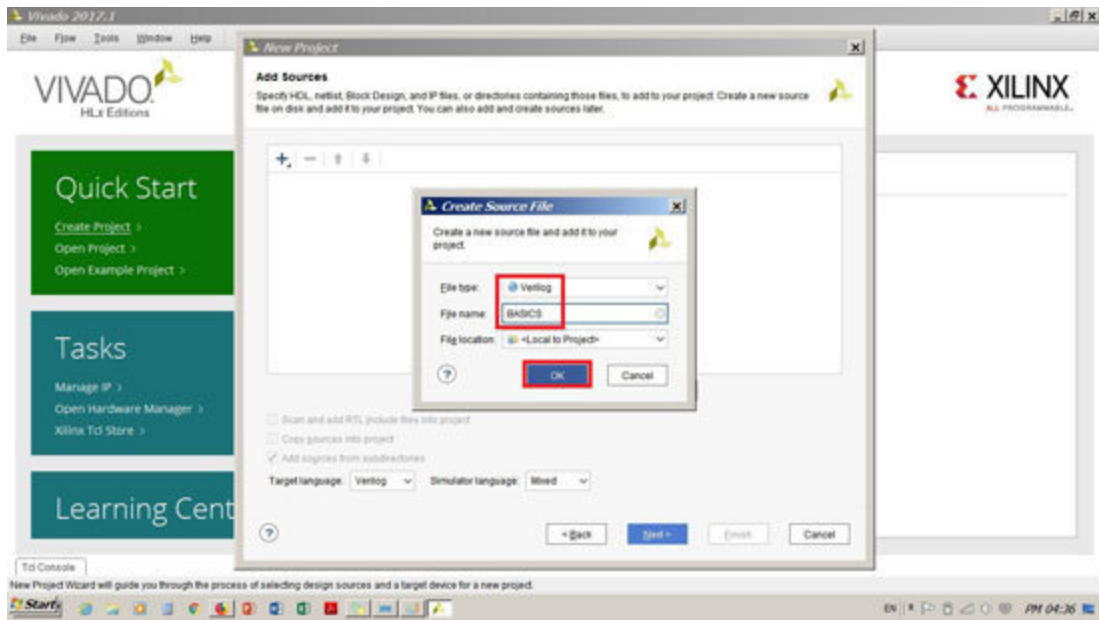


Figure 1.29: Selection of proper file type & enter the file name

Once the file type & file name are entered, the **New Project** wizard will look like [figure](#)

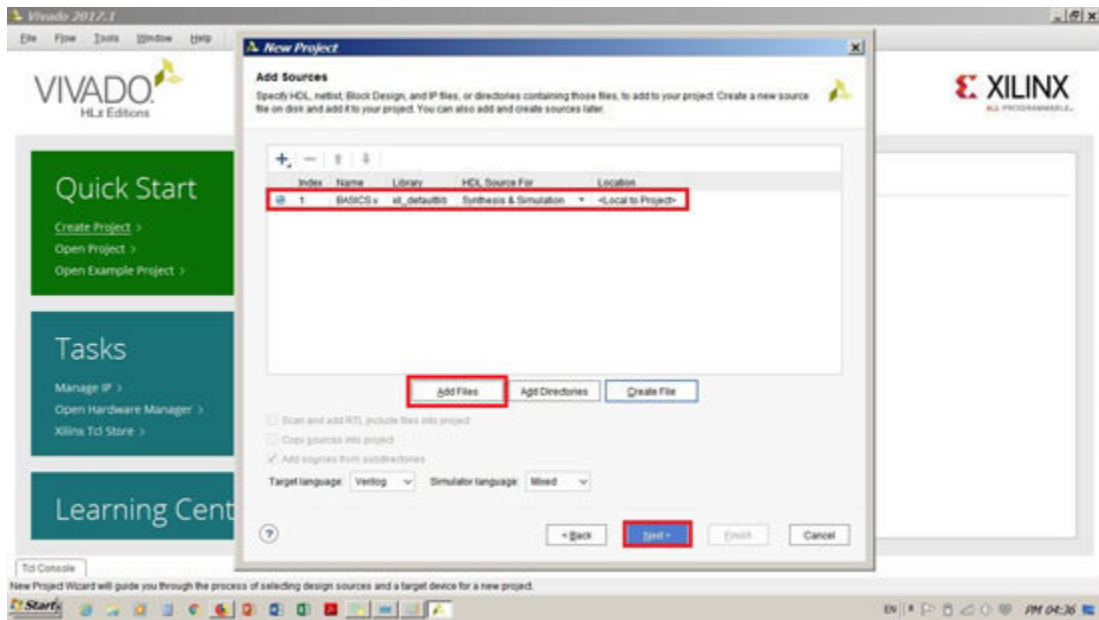


Figure 1.30: To add an existing source

Once the new file has been created, the same will be visible in the **Add Source** wizard. If someone wants to add a file (which is already existing in the library) can be added by clicking on the **Add files** button, as shown in [figure](#)

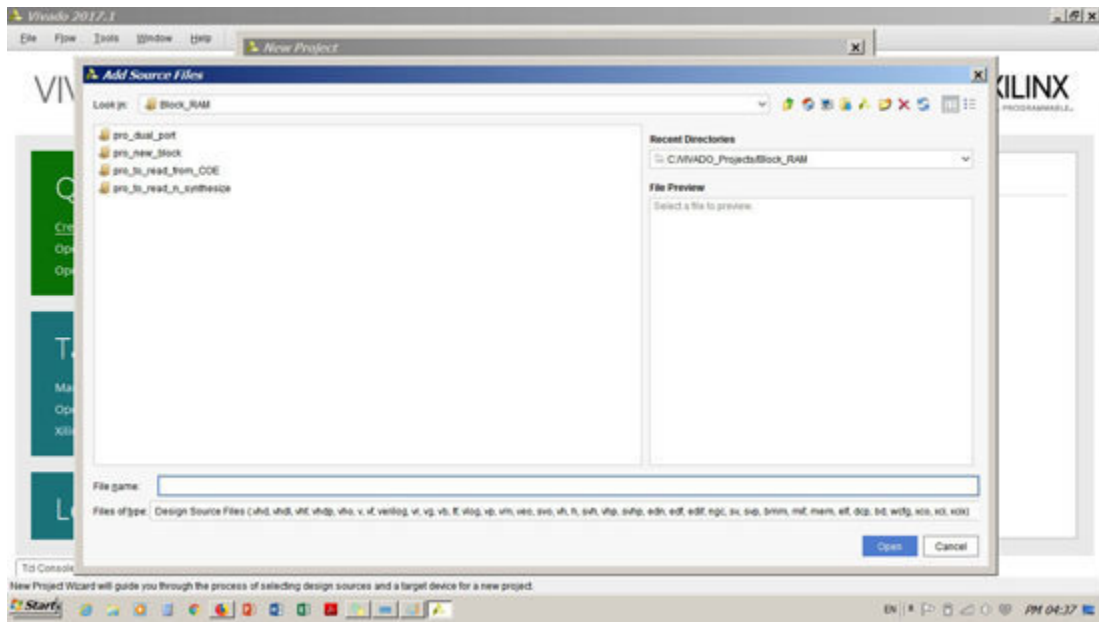


Figure 1.31: Window browser for selecting an existing source

Browse the existing source file & click on Open to add it to the project, as shown in [figure](#)

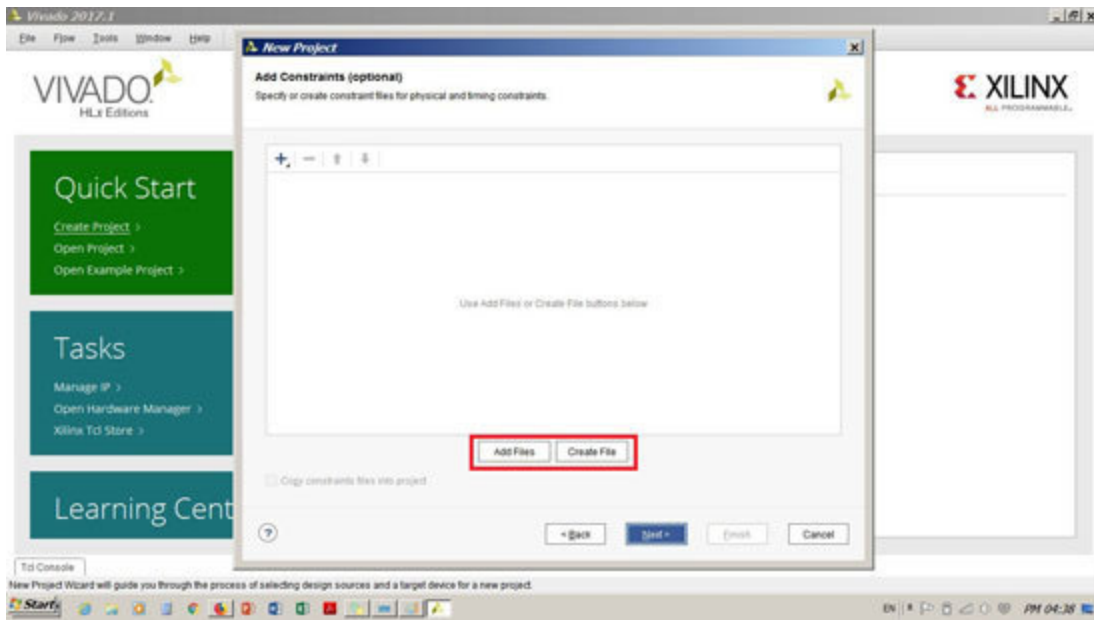


Figure 1.32: To add a constraint file

Adding a constraint file is optional. The constraint act as a bridge between the hardware of the FPGA board & the top module. The constraint file uses the extension as .xdc, which stands for the Xilinx design constraint file. For simulation purposes, there is no importance of the .xdc file. But or physical design on FPGA, the .xdc file is a must.

Choose the appropriate FPGA board available with you. Again, the selection of the exact board is not that important for logic simulation. But for synthesis & implementation on FPGA, the selection of an appropriate FPGA board is essential.

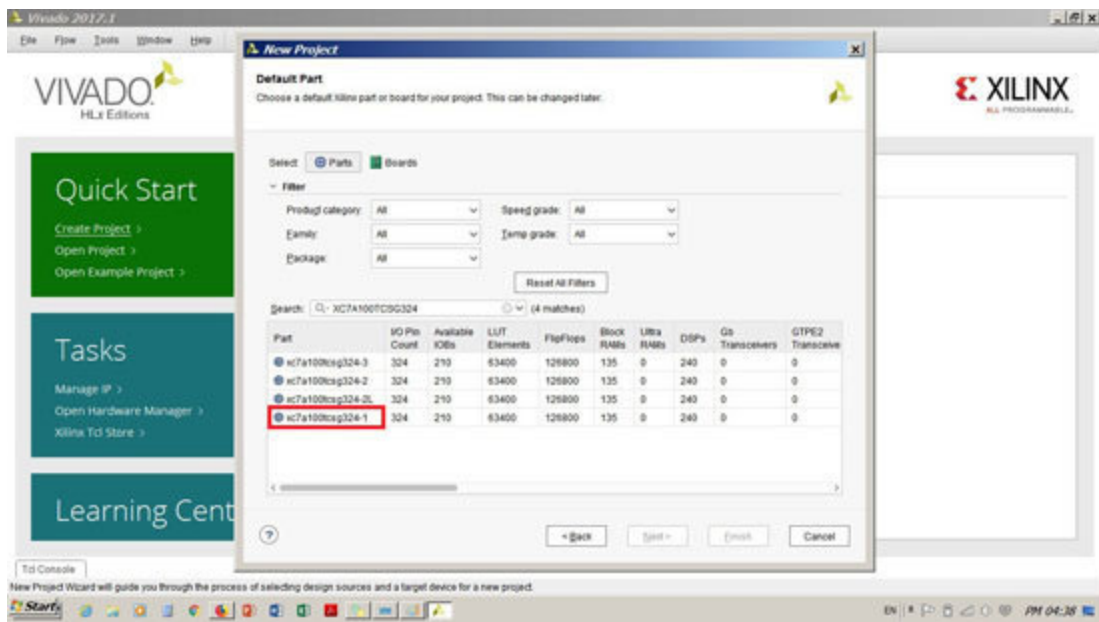


Figure 1.33: To select the appropriate FPGA board

Finally, the project summary will be displayed. If everything selected is proper, then click

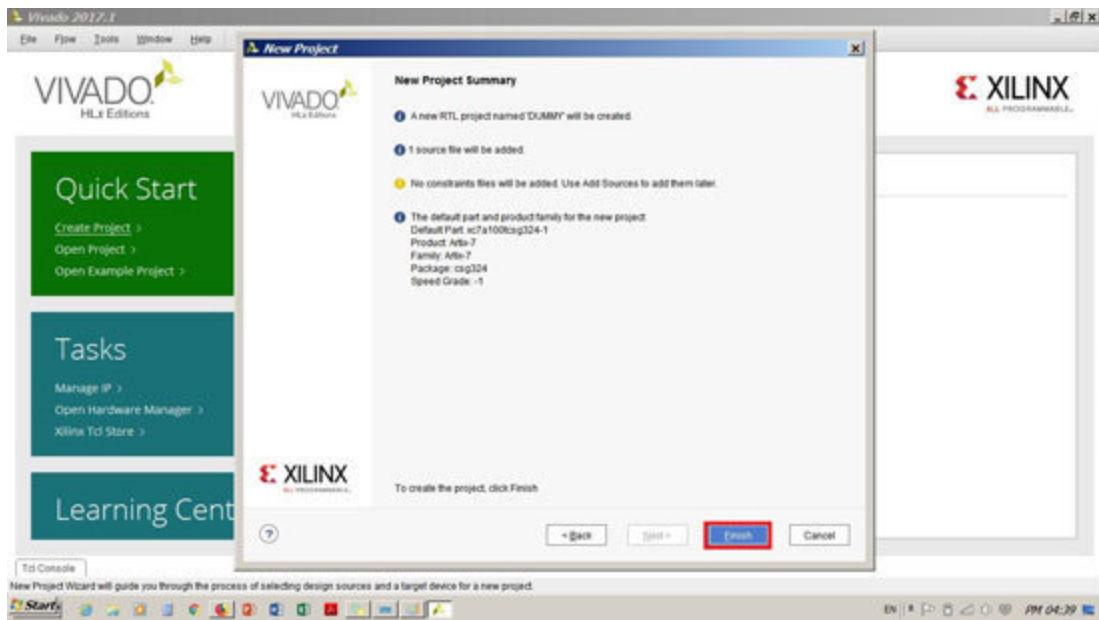


Figure 1.34: Summary of the new project

The wizard will take to initialize the new project.

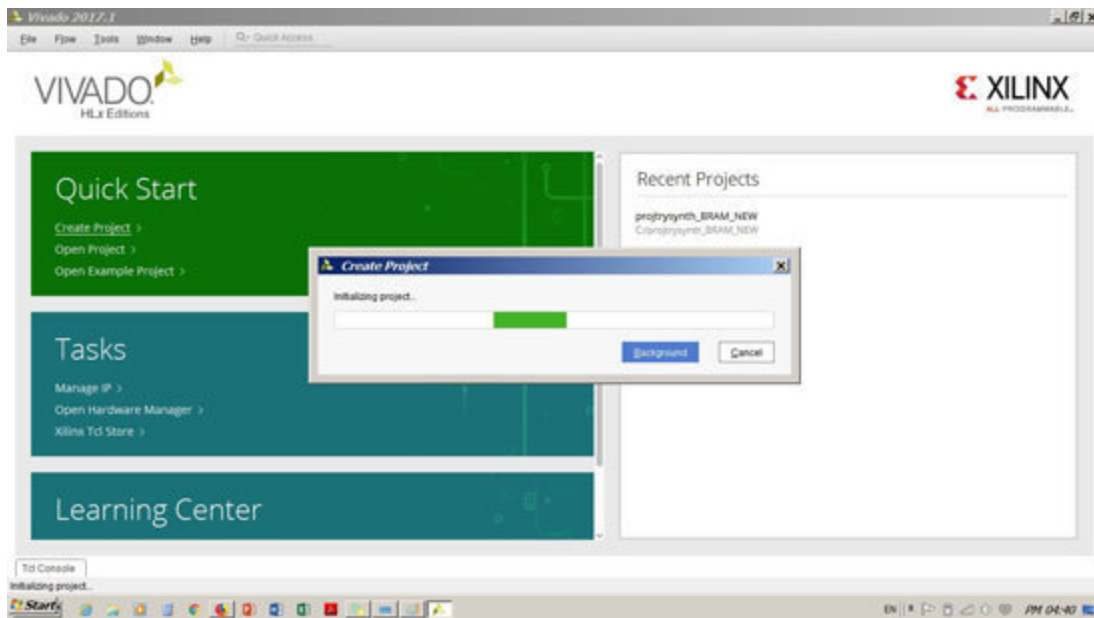


Figure 1.35: Initializing the new project

In Vivado, by default, the source file is taken as the module name (which can be changed later on). Therefore, before opening the editing window, the tool will ask for the module name & it's the port declaration. As shown in [figure](#) if changes are needed for the module name, do the same & provide the port names & their directions along with the bit size if the

ports are vector. Once completed, press the **OK** button to continue.

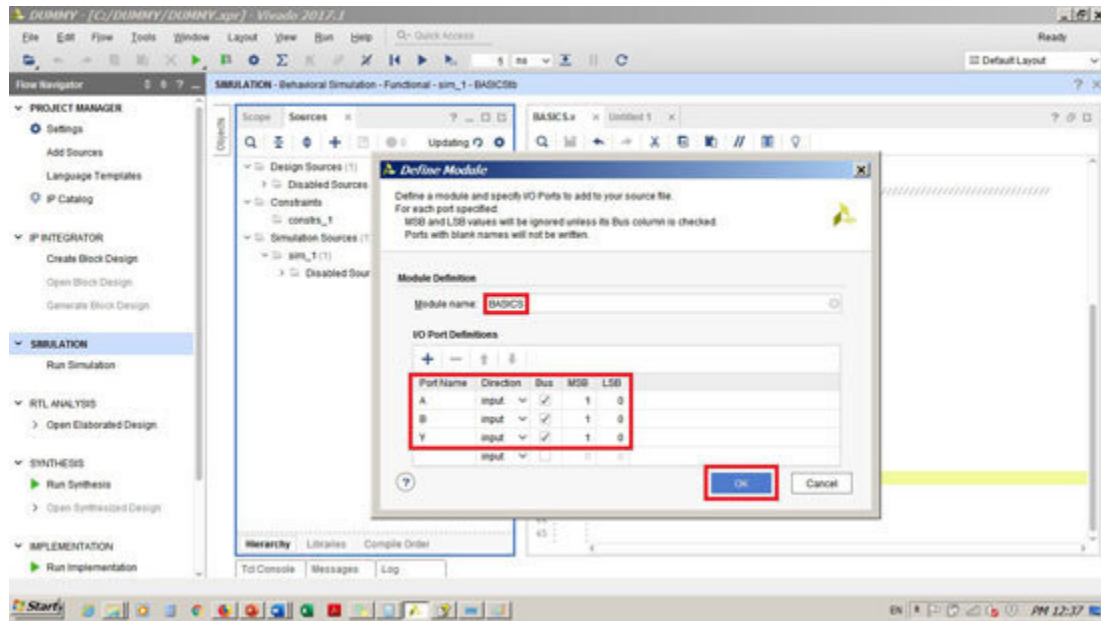


Figure 1.36: To provide the port list & directions to the new module

[Figure 1.37](#) shows the basic view of the tool. It consists of a **Project Manager Window**, **Source file Window**, **Project Summary Window** & **Status**. The **Project Manager Window** provides the selection of simulation, synthesis, implementation, bitstream generation, etc. of the source code. The **Source file Window** primarily provides three sources. They are:

Design source

Simulation source

Constraints

The design source provides the hierarchy of the modules for FPGA implementation. The simulation source provides the hierarchy of the modules for simulation. Whereas, the constraint provides the file to the design source. The **Status window** provides the syntax error & the other warnings to the programmer.

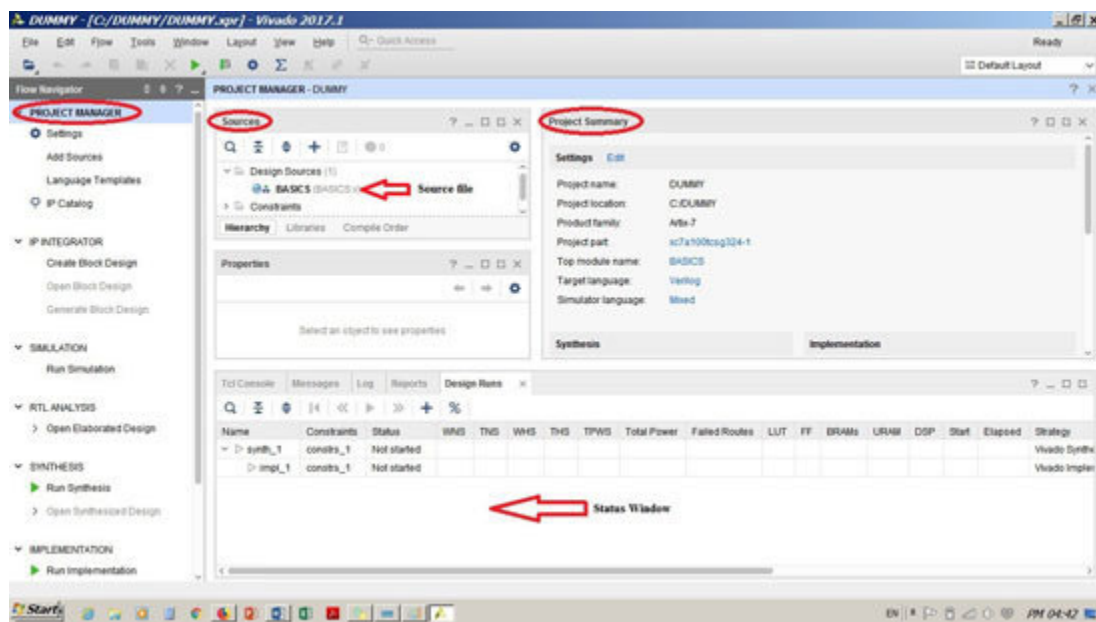


Figure 1.37: Different windows in Xilinx Vivado

Now, to update the content of the source file, right-click on the source file (inside the design/simulation source) & click on **Open** as shown in [figure](#)

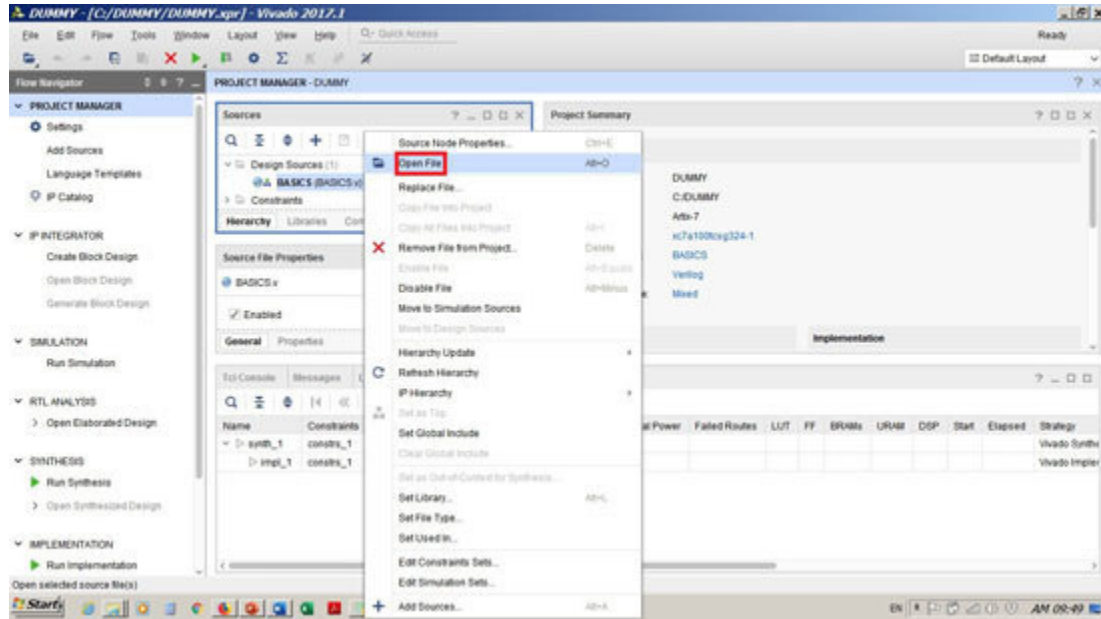


Figure 1.38: To open the newly added source

A new editor tab will open with the name the same as the source file name, as shown in [figure](#)

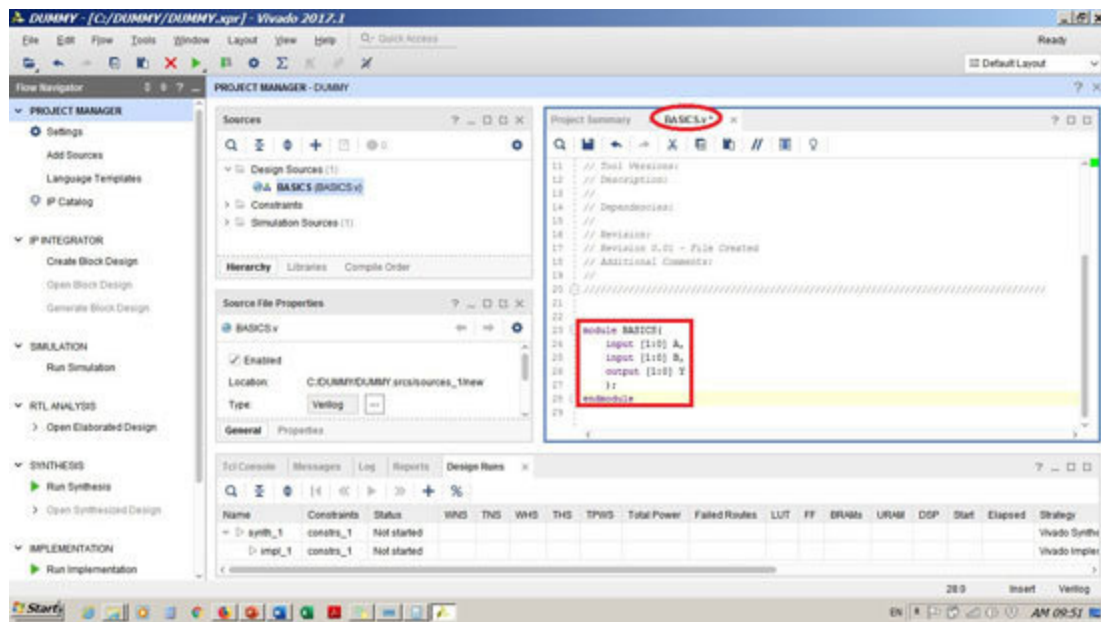


Figure 1.39: To edit the newly added source

Modify the program as per the need. If necessary, the test bench or the top module (in case of synthesis) can be typed in the same source file, as shown in [figure](#)

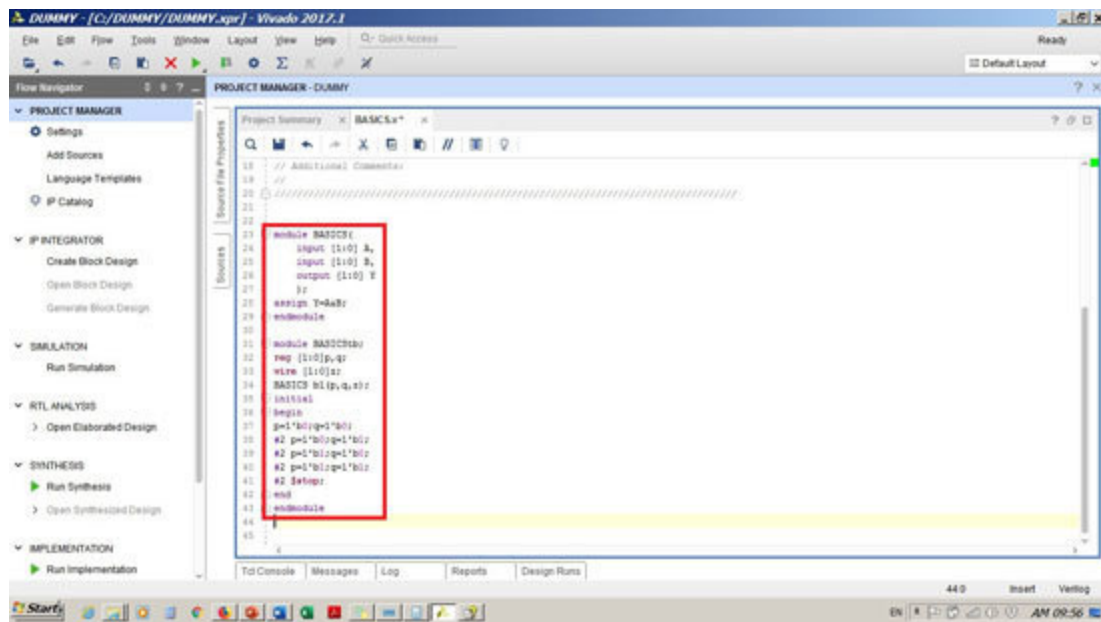


Figure 1.40: The modified source file

Once the program is typed along with the test bench, save the code by pressing Ctrl + S.

1.2.2. Simulation of project

After saving the code, open the Source Window & open the Simulation Sources to verify the hierarchy, i.e., the test bench module should be at the top & the main modules (or sub-modules if present) should be as per their instantiation order. The same is shown the [figure](#)

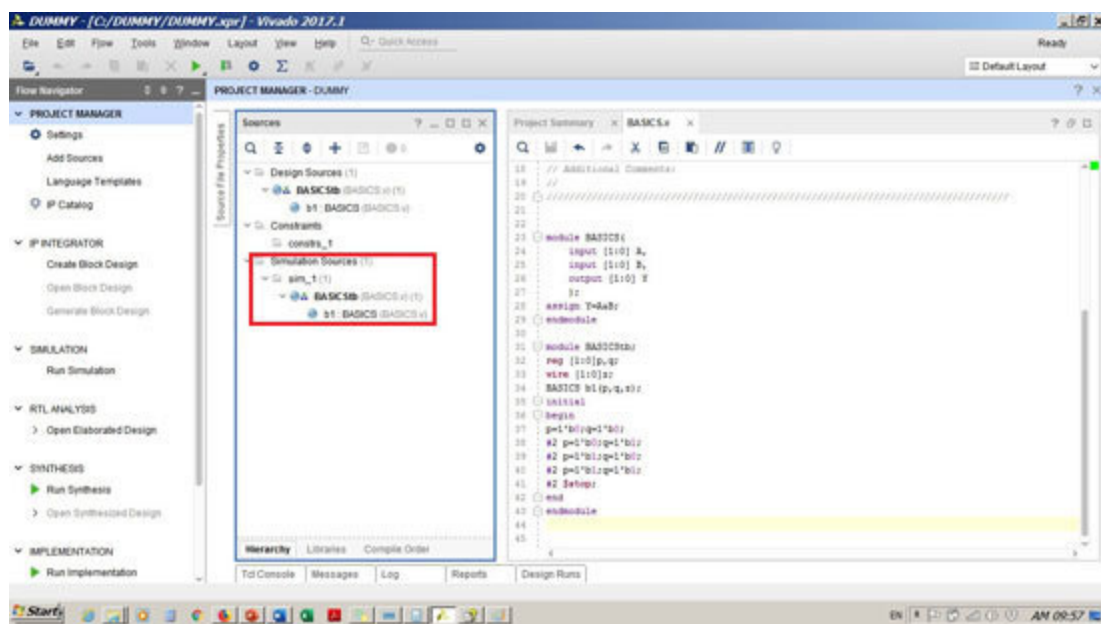


Figure 1.41: Expansion of Simulation Source

Now, in the **Project Manager** window, click on the **Run Simulation** → **Run Behavioural Simulation** button.

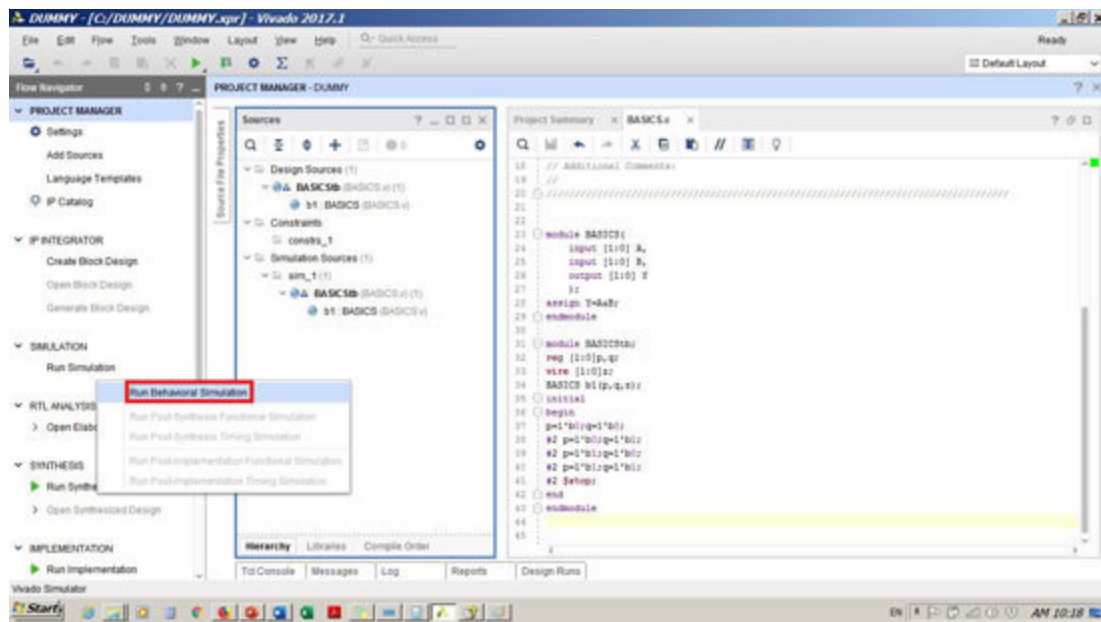
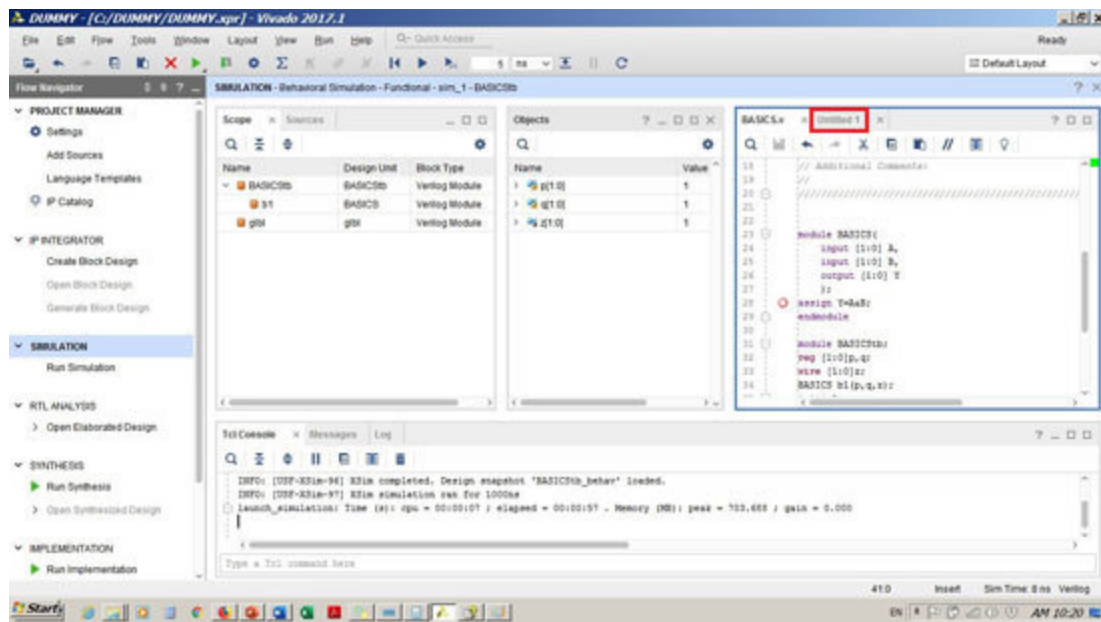


Figure 1.42: Expansion of Simulation Source

The system will take some time to execute the simulation & a new tab in the name of **untitled 1** will be generated in the **Project summary** window.



Click on the untitled 1 tab & maximize it, as shown in [figure](#)

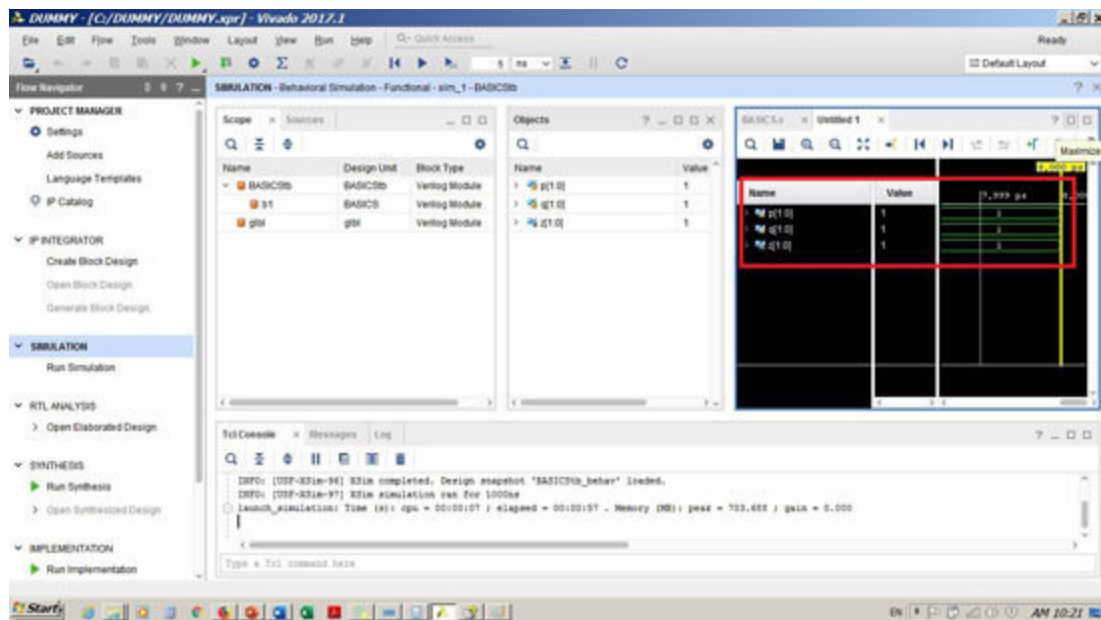


Figure 1.44: Waveform window in minimized mode

For getting a clear view of the waveform, click on the **Zoom fit** button, as shown in [figure](#)

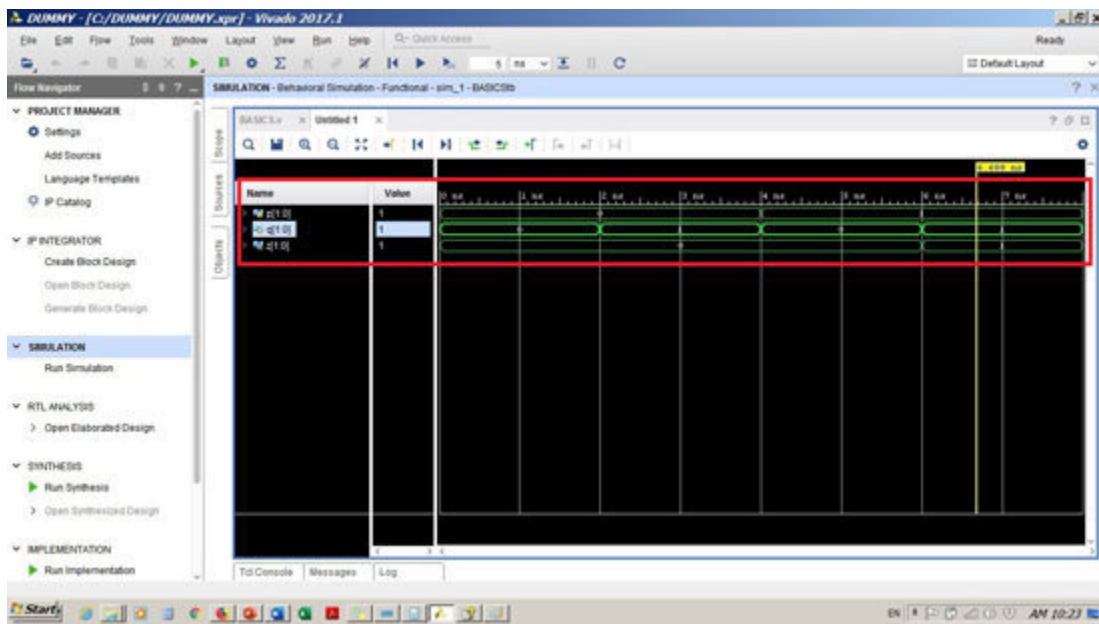


Figure 1.45: Waveform window in maximized mode

1.3 The Cadence NC-SIM

The merger of **Solomon Design Automation (SDA)** systems and ECAD, Inc. founded Cadence Design Systems, Inc. in 1988. During the tenure of first CEO Joseph Costello, Cadence became the most significant electronic design automation company producing software to design chips, PCBs, as well as IP cores for memories, SoC, interfaces, etc. [26]. Cadence products mainly focus on FPGA and ASIC designs for custom, analog, digital, and mixed-signal applications. Few of them are Encounter platform, Virtuoso Platform, Incisive Platform, Palladium Series, OrCad/PSpice, etc. SIM is an EDA environment in which the architectures are designed using any Hardware description languages (HDLs) such as Verilog HDL or VHDL. The followings are some of the essential points that one should consider before doing HDL coding on Cadence NCSIM:

Cadence works only on the LINUX operating system. Therefore, before starting with HDL, it is necessary to have a high-end computer in LINUX operating system with the Cadence bundle installed in it.

As LINUX systems work with command prompt, one should be aware of basic LINUX commands.

The step by step procedure for operating Cadence NCSIM is as follows:

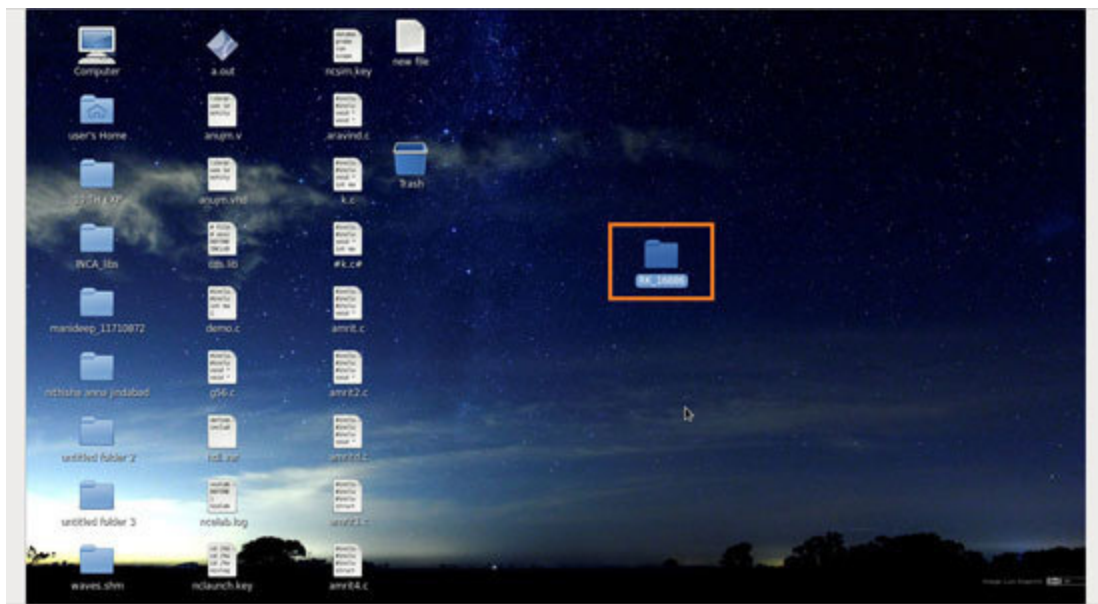


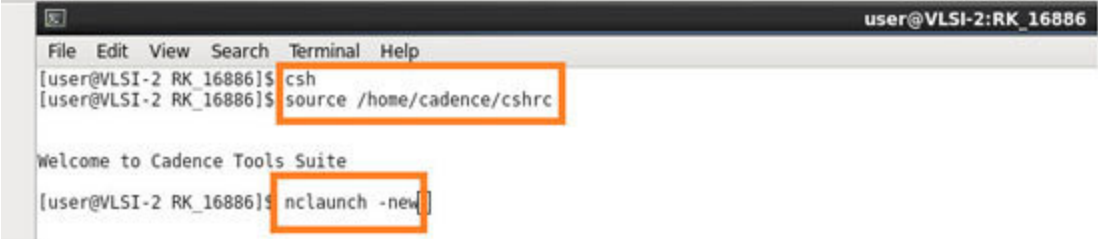
Figure 1.46: *Creating a folder in the Desktop*

To begin with, Cadence NCSIM, firstly create a folder at your Desktop. By right-clicking on the folder, click on the **Open in Terminal** option. A new terminal window will pop up (command prompt). In this terminal window type the following commands as shown in [figure](#)

ssh

source /home/cadence/cshrc

nclaunch -new



The image shows a terminal window with a menu bar (File, Edit, View, Search, Terminal, Help) and a title bar (user@VLSI-2:RK_16886). The terminal content shows the following sequence of commands and output:

```
[user@VLSI-2 RK_16886]$ ssh
[user@VLSI-2 RK_16886]$ source /home/cadence/cshrc

Welcome to Cadence Tools Suite
[user@VLSI-2 RK_16886]$ nclaunch -new
```

Three specific parts of the terminal output are highlighted with orange boxes: the `ssh` command, the `source /home/cadence/cshrc` command, and the `nclaunch -new` command.

Figure 1.47: Command window

1.3.1. Creating a new project

A window with different selection modes will open up. For running the program in a single go, click on the **Single Step** option else click on the **Multiple Step** option. But in our case, we have considered the **Multiple Step** option. [Figure 1.48](#) shows the NCSIM Run mode.

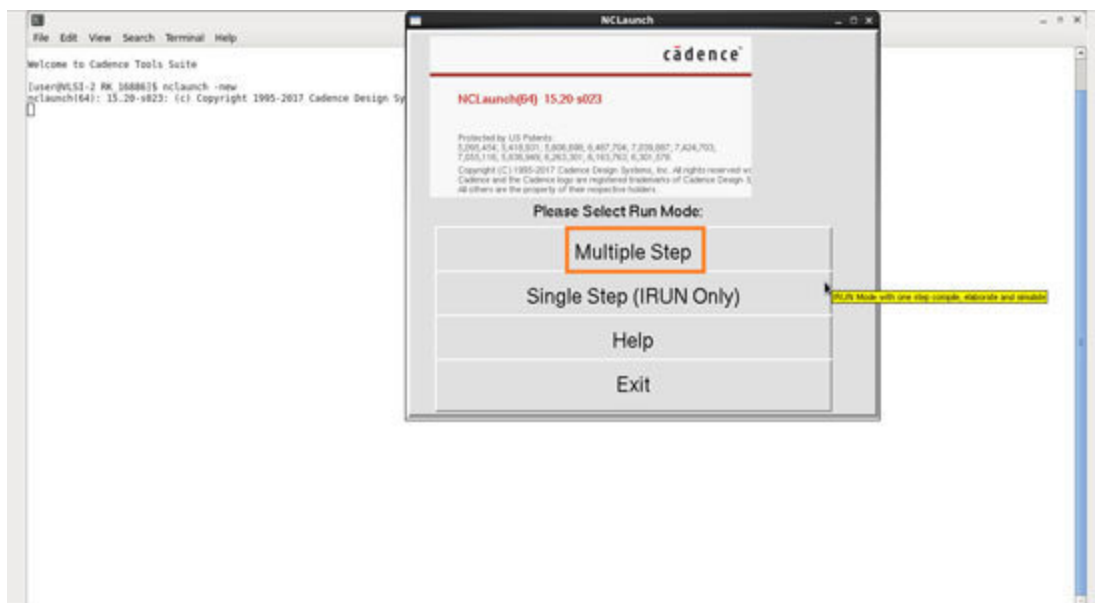


Figure 1.48: Selection of NCSIM Run mode

Upon clicking on the **Multiple Step** option, a new window will pop up for creating a cds.lib file, as shown in [figure](#). This

cds.lib file defines the reference libraries common to all your projects, and a cds.lib in your project directory also defines the project-related libraries. Moreover, for opening the tool for the first time only requires to create a new cds.lib file. Therefore, this step can be skipped while opening the tool a second time onwards.

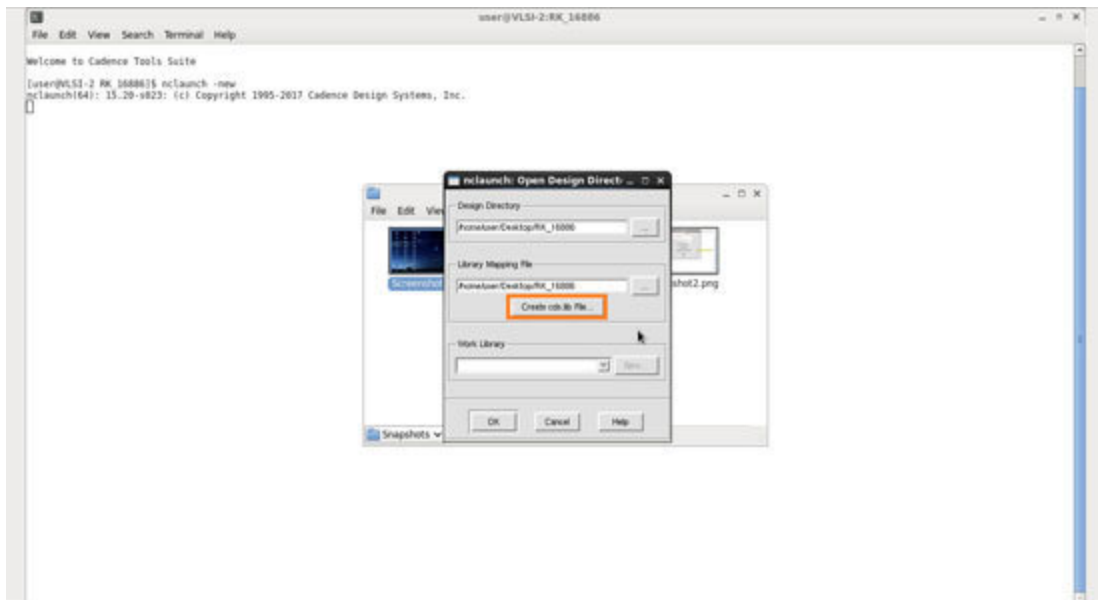


Figure 1.49: *Creating a cds.lib file*

Now to create a new cds.lib file, click on the **Create cds.lib file** button, as shown in [figure](#). The parent folder will open up (which was created at the Desktop earlier). Click on the **Save** button, as shown in [figure](#).

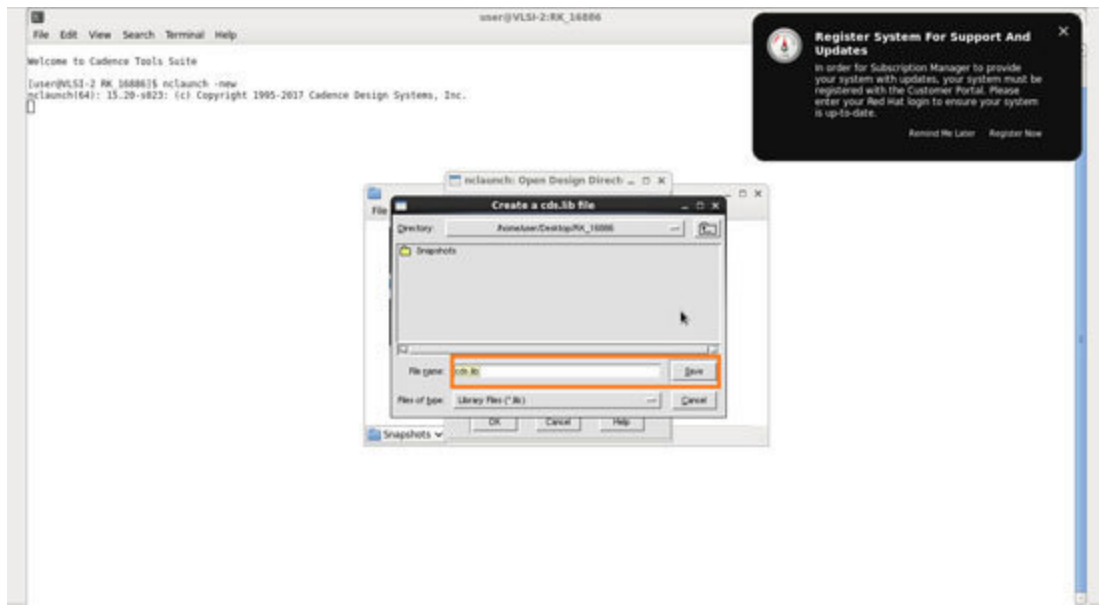


Figure 1.50: Saving a *cds.lib* file in the parent directory

In the **New cds.lib** File wizard chooses the correct option as per the HDL used for programming. For VHDL, one may choose either **Include default library** or **Include IEEE pure library**. On the other hand, for Verilog HDL, choose the 3rd option, i.e., **Don't include any libraries**.

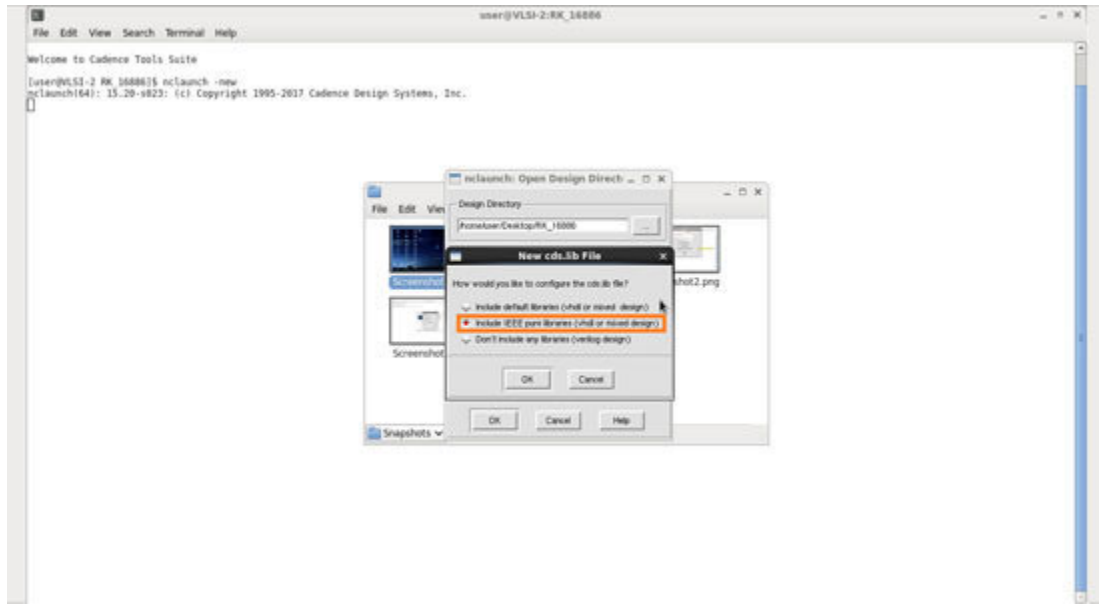


Figure 1.51: Configuration of cds.lib file

After making appropriate selection click on the **OK** button & finally in the **Open Design Directory** wizard press **OK** to continue as shown in [figure](#). The cds.lib file must be generated inside the parent directory only.

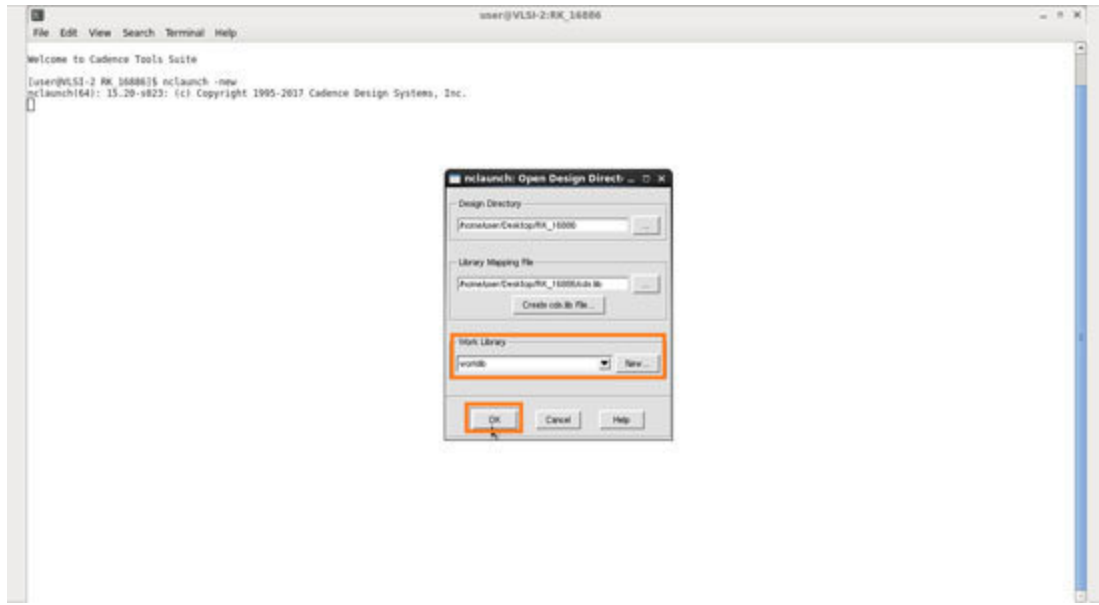


Figure 1.52: Final selection of design directory, library mapping file & work library

After the creation of the cds.lib file, the tool will open up. This window consists of 3 sub-windows, namely source file window, worklib & snapshot window & status window. It is shown in [figure](#)

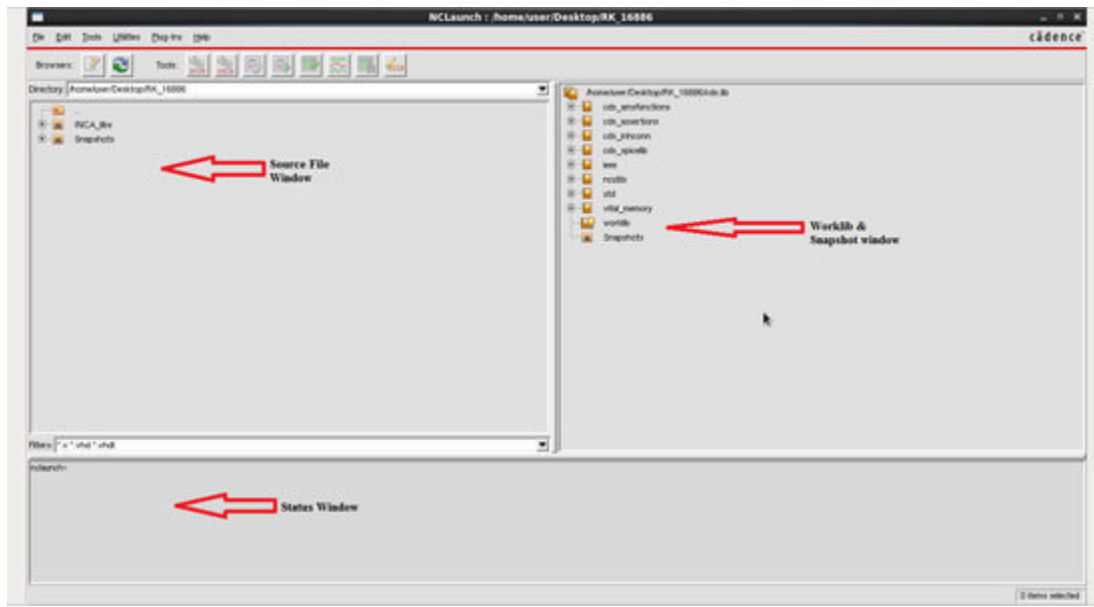


Figure 1.53: Sub windows in the NCSIM tool

The source file window consists of the HDL source files. The extension for the Verilog HDL file is .v & the extension for VHDL file is .vhd. The status window shows the status of the program. i.e., the errors, warnings, etc. are displayed on this window. And lastly, the worklib & snapshot window consists of the elaborated files. These files are finally used for running the simulation.

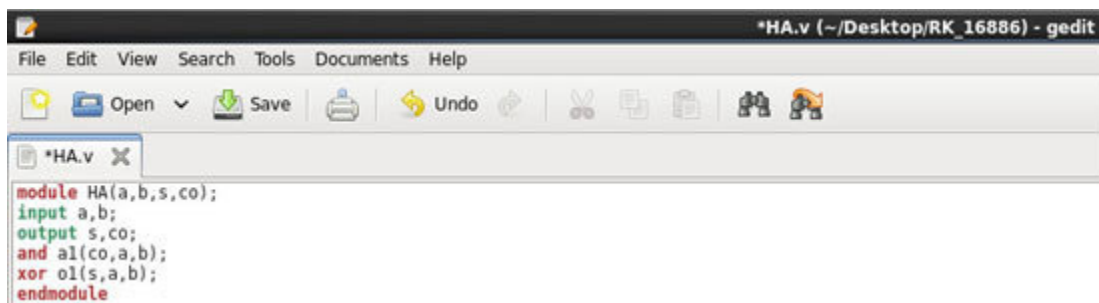


Figure 1.54: Main module in a text file with .v extension

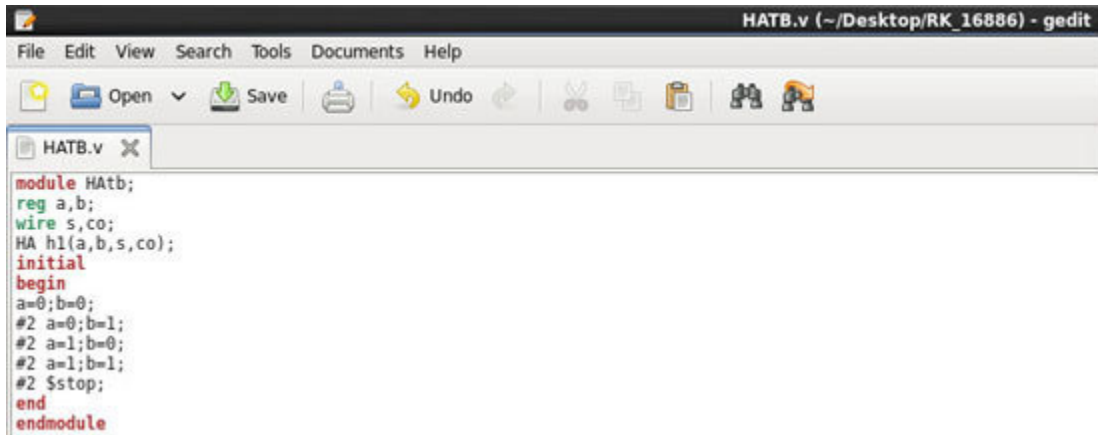


Figure 1.55: Test bench module in a text file with .v extension

Now, to start typing the HDL code, go to the parent folder/directory (which was created at the Desktop earlier) & right-click inside the folder & click on **Create document** → **Empty file**. This document file should have an extension of the desired HDL file; otherwise, these source files will not be detected in the source file window. Therefore, rename the text file with the desired file name along with proper extension (i.e., .v for Verilog HDL & .vhl for VHDL). You may create multiple files for writing the main module, sub-modules & test bench. But a single file can also handle multiple modules as well. In this example, we have created two source files (one for the main module & the other one for the test bench), as shown in [figures 1.54](#) and

1.3.2. Simulating the project

Immediately after saving the content of the file, the name of the source file will be visible in the source file window. In this example, two source files, namely HA.v & HATB.v, which consists of the main module & test bench respectively, are created & accordingly, both are visible in the source file window, as shown in [figure](#)

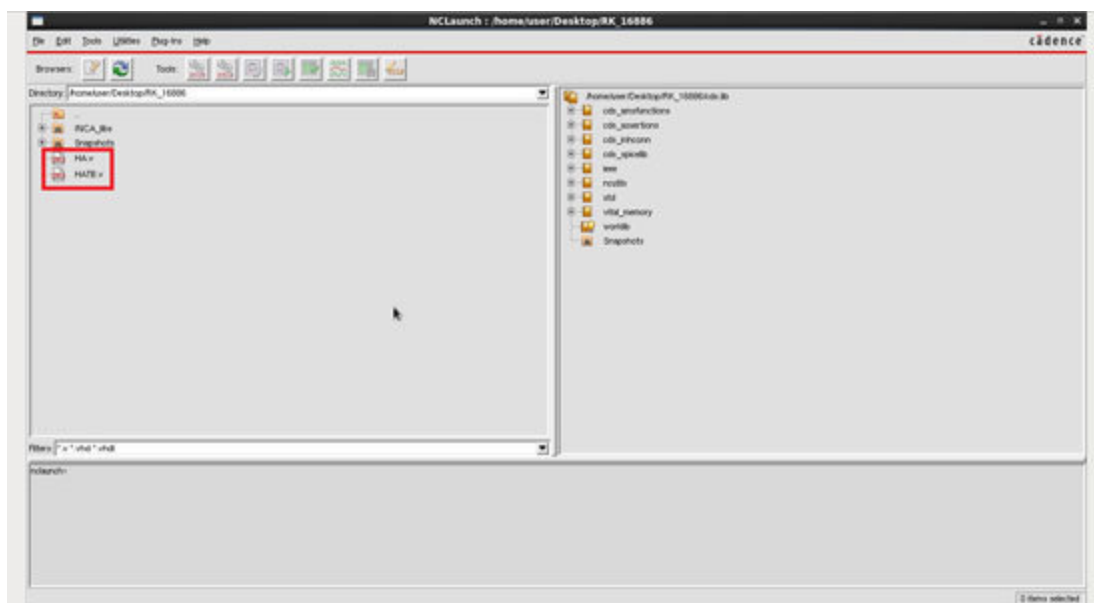


Figure 1.56: Appearance of the source files in the source file window

Once the source files are added, the next step is to check syntax. For doing the same double click on the main module source file. The status window will show the current status of the code. If there is an error, revisit the code once again & check the syntax of the code once again by double-clicking the main module source file.

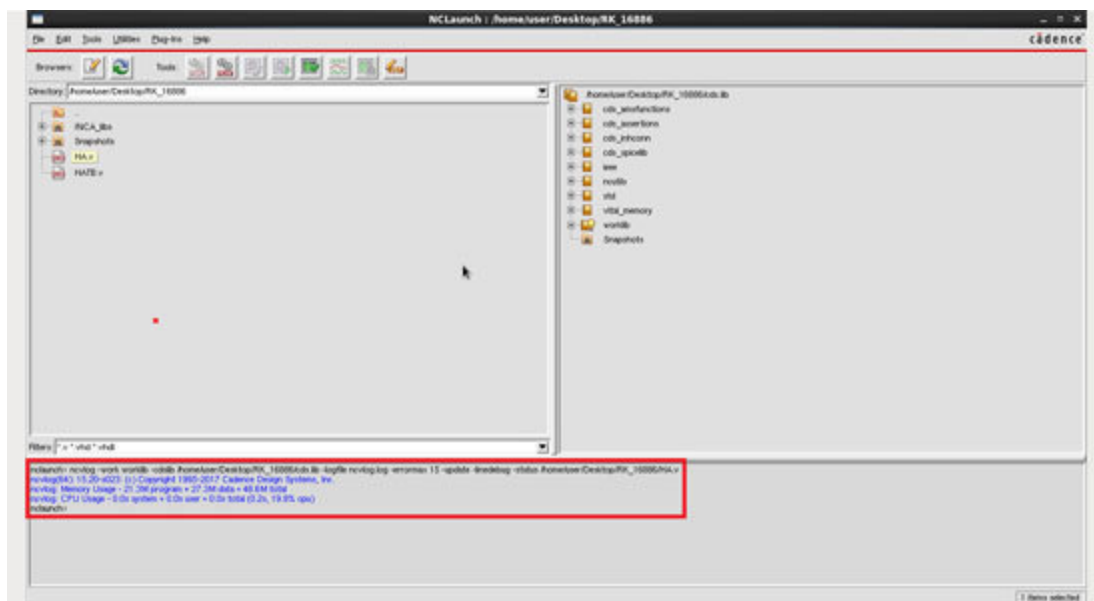


Figure 1.57: Status window after running the check syntax for the main module

Repeat the same process for the test bench source file as well. For the user reference, the snapshots of the status window are attached in [figure 1.57](#) and 1.58.

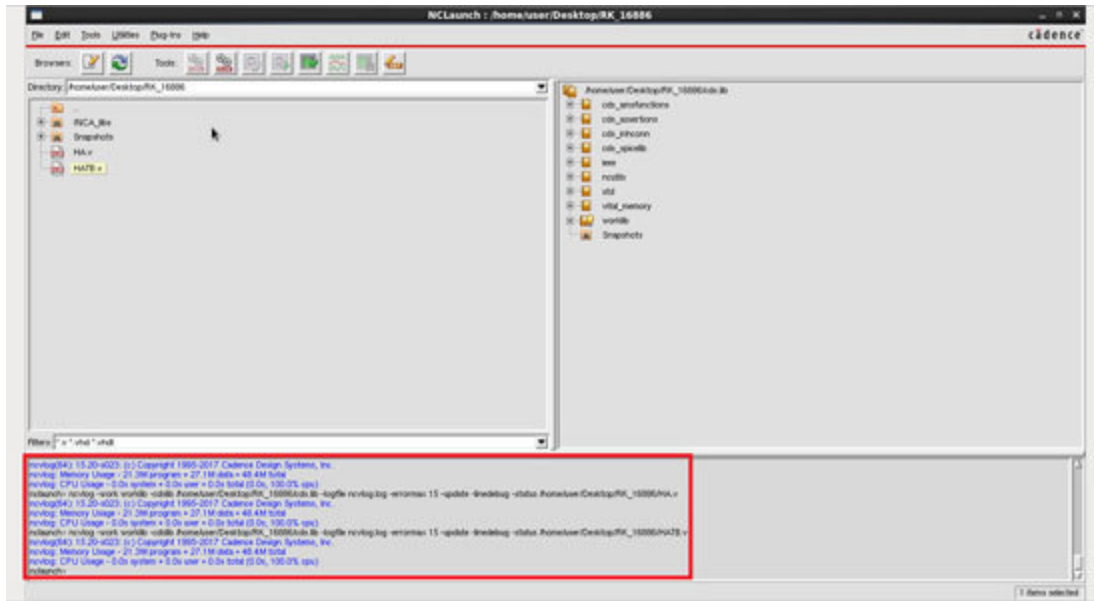


Figure 1.58: Status window after running the check syntax for the Test bench module

The next step is to elaborate on the code. For doing the same, go to the worklib folder available inside the worklib & snapshot window. Inside the worklib folder, the main module name, submodule names & the test bench name will be visible

Here, not the file names, but the module names will be visible.

Therefore, inside a single source file, if we have multiple modules, then all module names will be visible inside the worklib folder, as shown in [figure](#)

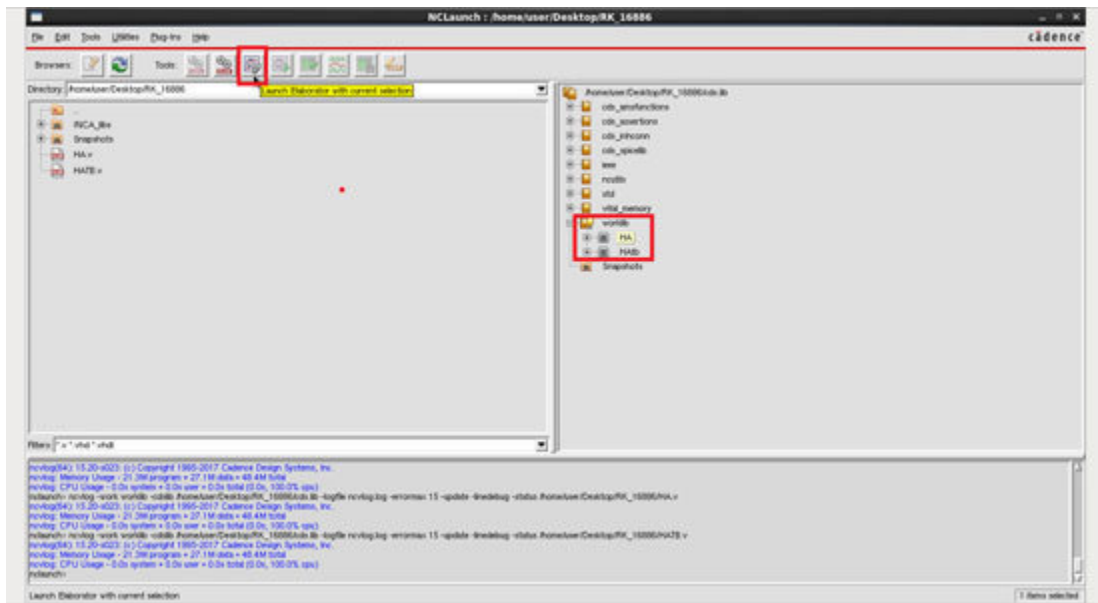


Figure 1.59: Status window after running the elaboration for the main module & test bench module

To elaborate the code, firstly click on the main module & click on the **Elaborate** button available at the top panel, as shown in [figure](#). If there is/are any elaboration errors in the code, the same will be notified in the status window. Once elaboration of the main module is completed, follow the same procedure for the test bench module. In design, if we have multiple designs, then (ideally) elaborate the bottom-most module first & then keep on elaborating the rest of the modules as per the hierarchy from bottom to top. In the next step, launch the simulator by clicking on the test bench worklib file available inside the snapshot folder under the worklib & snapshot

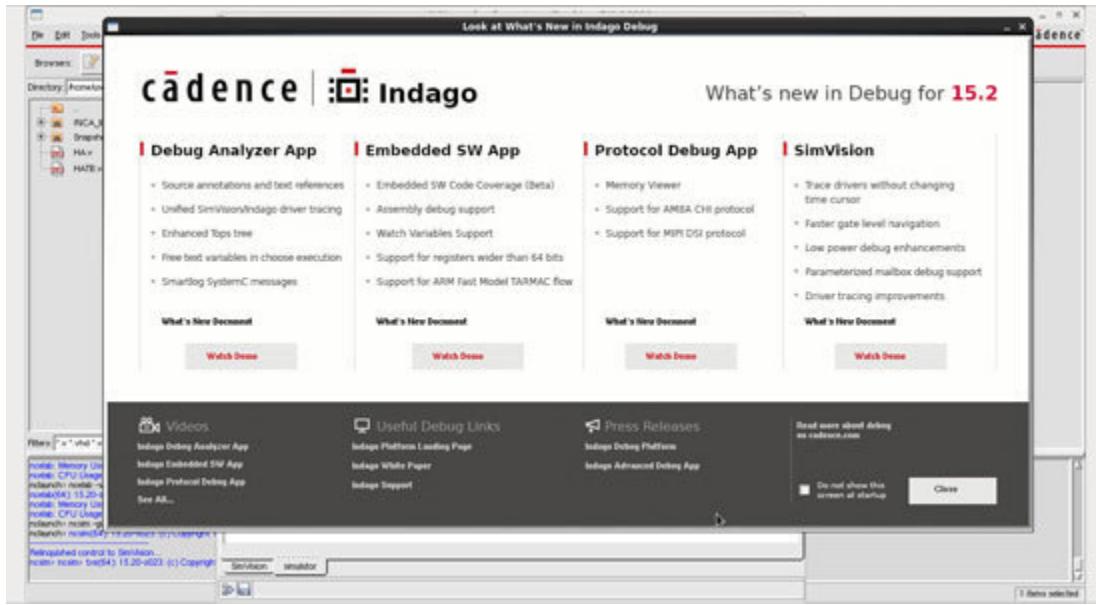


Figure 1.61: Launch screen of the NCSIM simulator

The simulation wizard consists of the **Design Browser & Console** window, as shown in [figures 1.62](#) and respectively.

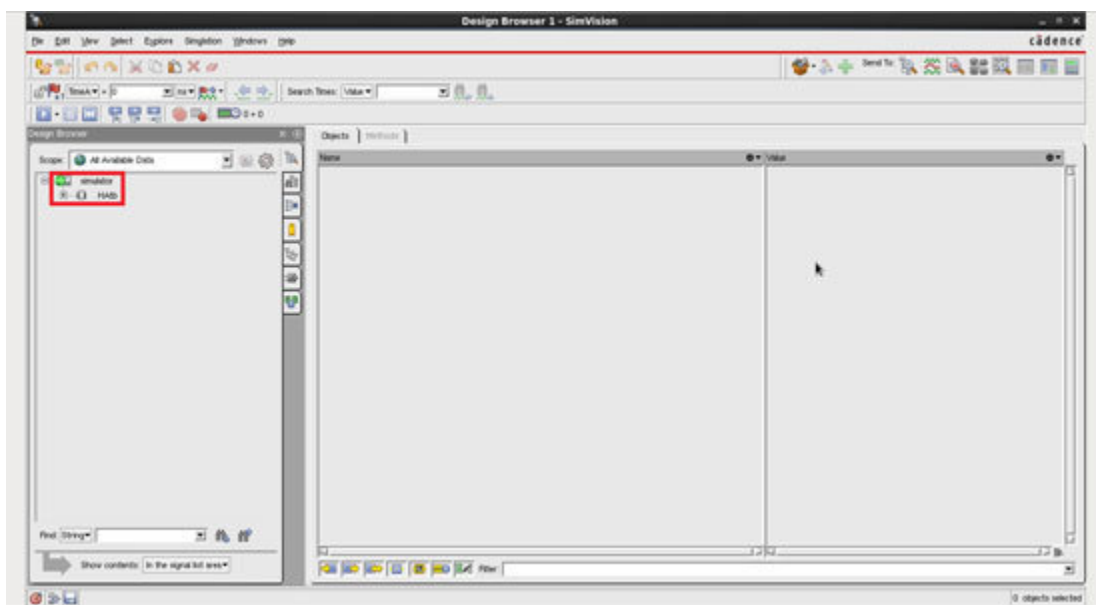


Figure 1.62: Design Browser of the NCSIM simulator

Once the simulator gets open, go to the **Design Browser** and click on the test bench module name, which will appear at the top left-hand corner.

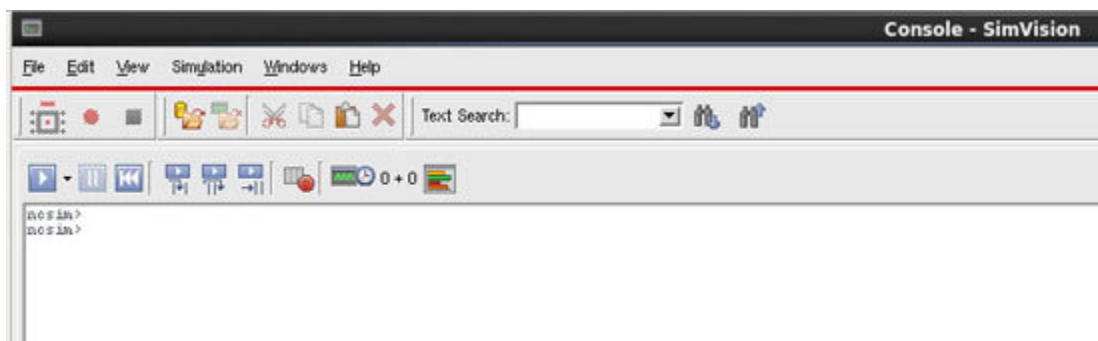


Figure 1.63: Console window of the NCSIM simulator

Upon clicking the test bench module in the design browser, the list of local objects (i.e., test bench's registers & wires) will be visible. Select all of them & click on the **Send selected objects to target the waveform window** button.

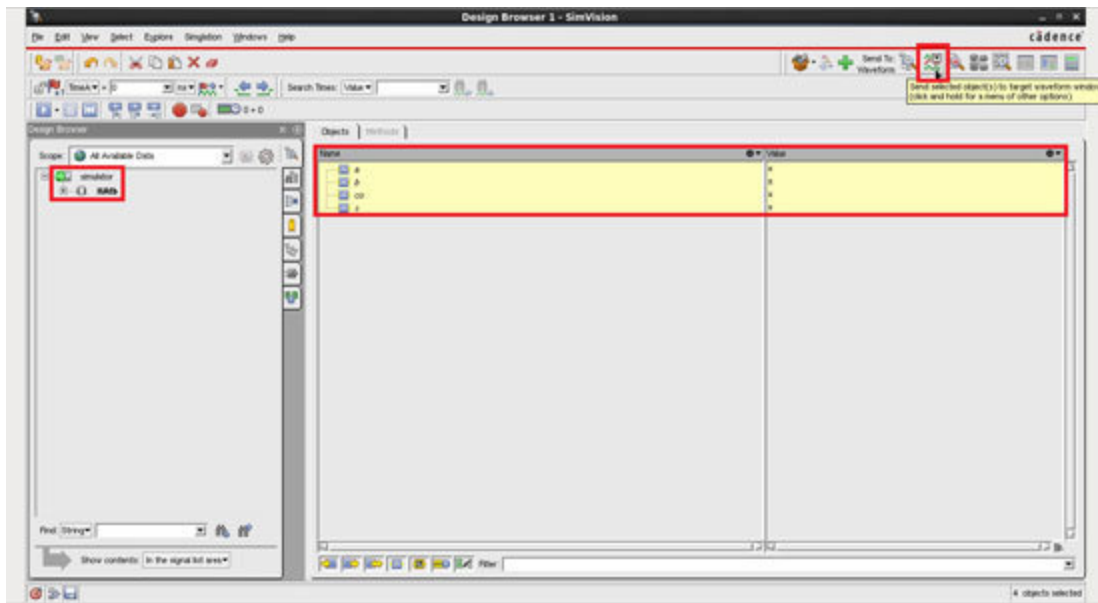


Figure 1.64: Design Browser with the list of local objects

Using the above command, waveform window will open, where a list of all selected objects of the test bench will be visible. Now select the local objects of the test bench (all of them) & click on the **Run the simulation** button as shown in [figure](#)

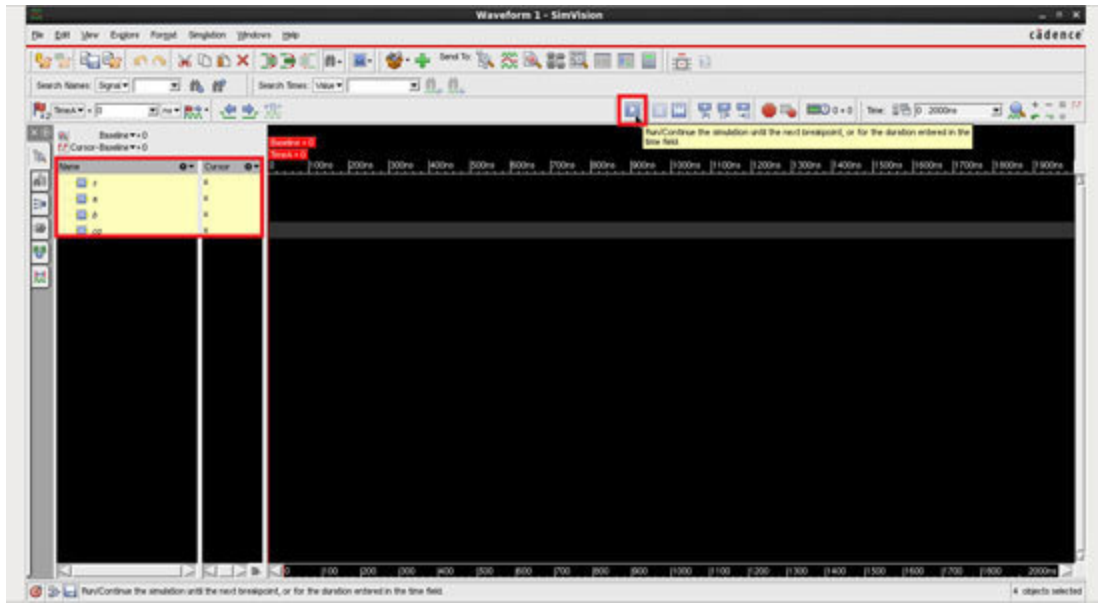


Figure 1.65: Waveform window of the NCSIM simulator

The waveform will be displayed, as shown in [figure](#)

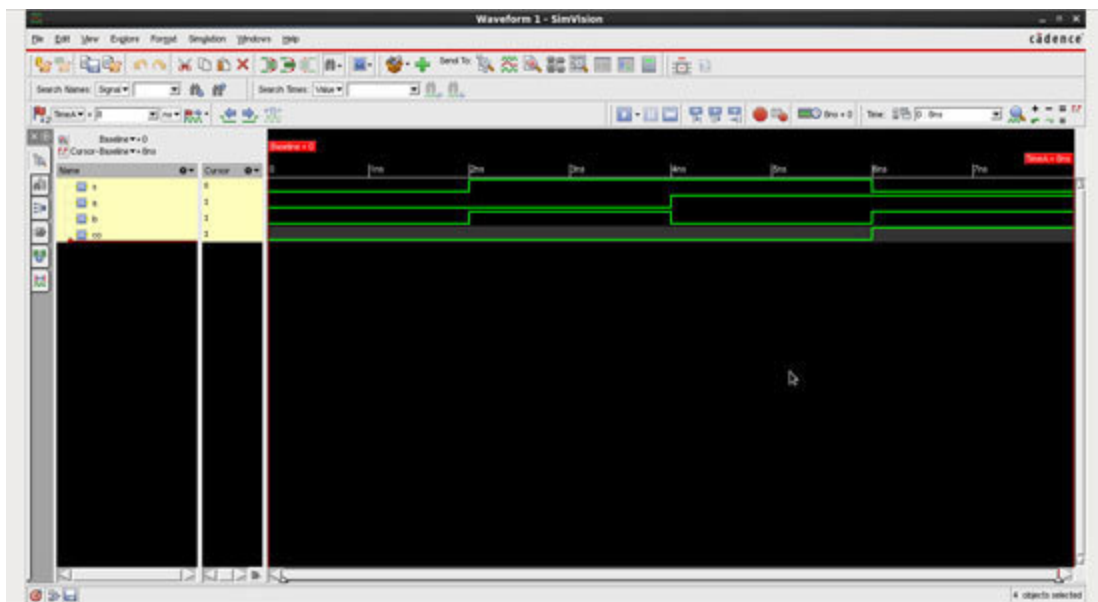


Figure 1.66: Waveforms with the list of local objects

Similarly, the RTL view of the design can be made available from **Design Browser** by choosing all local objects of the test bench module & clicking on **Send selected objects to target schematic** as shown in [figure](#)

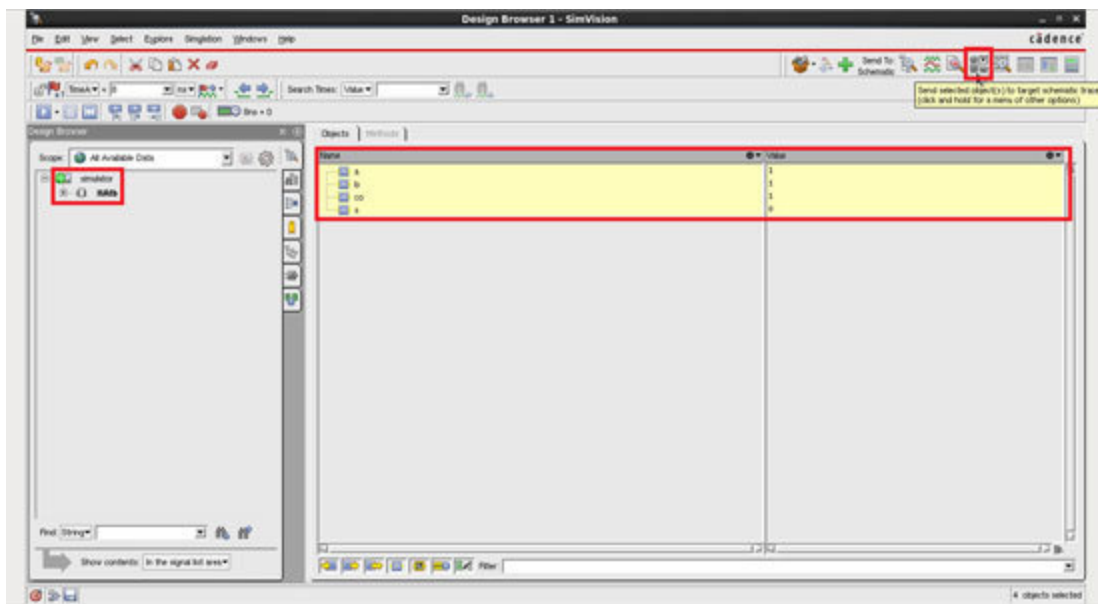


Figure 1.67: Design Browser for target schematic trace with the list of local objects

[Figure 1.68](#) shows the RTL view of the design.

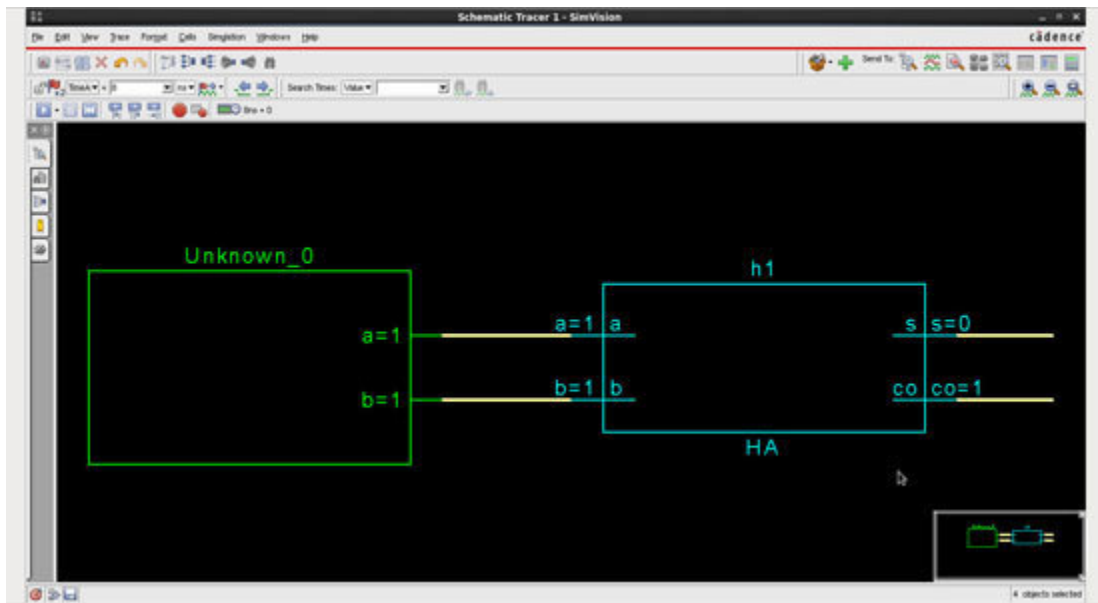


Figure 1.68: View of the schematic tracer

One can select the main module instance (in blue color) to see the local instances, as shown in [figure](#)

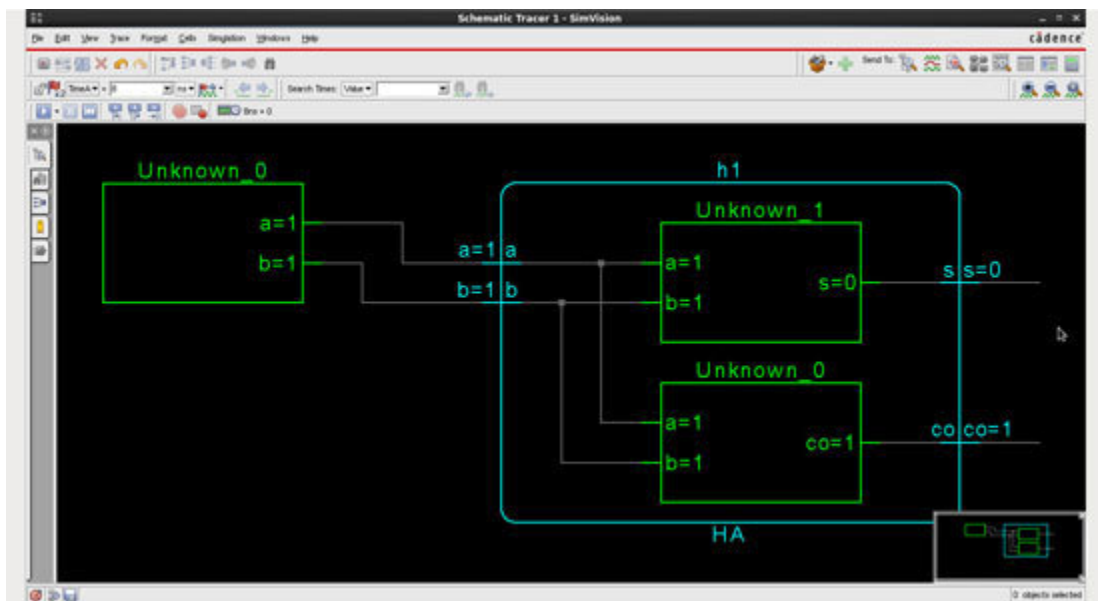


Figure 1.69: *View of the schematic tracer with an expanded view*

Conclusion

In this chapter, we studied different CAD tools that we use for HDL based project design. The design process right from project creation to RTL design along with simulation waveform generation is discussed in this chapter.

In the next chapter, we will study the need for Hardware Description Language. The brief introduction about Verilog HDL, along with its various levels of abstraction, will be explained with the help of programming examples in the next chapter.

Questions

What is ISE in Xilinx ISE stands for?

List any five EDA tool which is widely used across the globe.

What is the use of .xdc file digital designs?

During the configuration of cds.lib file in Cadence NCSIM,
Don't include any libraries is chosen in which HDL & why?

CHAPTER 2

The Need for Hardware Description Language (HDL)

2.1. Introduction to HDL for digital circuits

The structure and behavior of electronic circuits are described by HDL. It is a hardware description language that allows analysis and simulation of digital logic and circuits. An IC is created using synthesis, netlist generation, masking, and optimization. The HDL is an integral part of the **EDA (electronic design automation)** tool for PLDs, microprocessors, and ASICs. The hardware description language was created to implement RTL (register transfer level) abstraction. The HDL is similar to language C. The first HDL was designed in the late 1960s. The HDL involves a testbench, which is designed to verify the correctness of the circuit. The HDL proves to be an excellent language for CPLD and FPGA. HDL includes a textual description consisting of operators, expressions, statements, inputs, and outputs. The widely used hardware description languages are Verilog, Very high-speed integrated circuit HDL (VHDL), System Verilog, Verilog C. With the advancement of technology, the language is improved at an accelerated rate. The latest iteration of Verilog, IEEE 1800-2005 System Verilog has improved features and architectures for design hierarchy and reuse.

Structure

In this chapter, we will discuss the following topics:

Importance and need of Hardware Description Language

Digital system design using Verilog HDL

Modeling styles in Verilog HDL

Objectives

After studying this unit, you should be able to:

Design a digital system using a hardware description language

Modeling styles such as gate level, data flow, behavioral, and switch level modeling using examples.

2.1.1. Need for Hardware Description Language (HDL)

Initially, in the late 1970s, the designers work on PCBs or board. As per Moore's law, the number of transistors is doubled, after every two years. So, it was difficult for the designer to manage a large number of components using PCB and breadboards. The HDL was invented to facilitate the designer with computer-based circuit designing, optimization, and verification. The applications for HDL include:

Fast design and better verification are possible, using HDL.

Complex digital systems need more time for simulation, verification, synthesis, and debugging., these tasks are done simultaneously using HDL, which will save time. The sequential execution of statements is possible in HDL.

The different parameters, such as power, delay, and area, can be extracted using HDL.

Using HDL, the designer can make the necessary engineering trade-offs and can develop the design in a better and efficient way.

The testing of large circuits was not possible because the large circuits contain lots of gates, and the response of every gate could not be checked. HDL has evolved the testing process easy and feasible.

The designer can develop HDL codes on hardware using PLDs, which saves time as well as cost.

The Top-down design and hierarchical design method allow the design time, design cost, and design errors to be reduced.

HDL provides the timing information and allows the design to be described in the gate level and register transfer level.

HDL allows reusability of resources. Design reuse" is the process of migrating high-quality intellectual property (IP) from one ASIC design to another. Designs can be reused for variable-width implementations using unconstrained arrays.

Using HDL, the designers can create tools that manipulate the design for synthesis, verification, and optimization, etc.

So, for chip designers, HDL proves to be a significant boon. It facilitates the designer in terms of time, optimization, power,

area, and cost.

2.2. VLSI Design flow

The design of integrated circuits is shown in [figure](#). It depicts the process flow from design specifications to post-layout simulation.

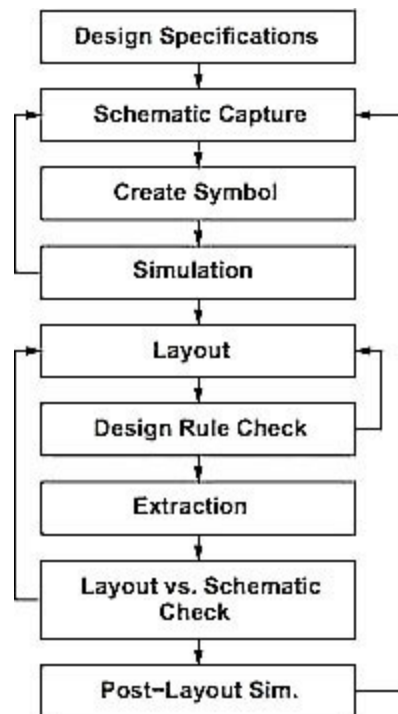


Figure 2.1: The typical VLSI design flow

2.2.1. Digital system design using Verilog HDL

Verilog, standardized as IEEE 1364, is a **hardware description language (HDL)** used to model electronic systems. It is most commonly used in the design and verification of digital circuits at the RTL level of abstraction. Verilog is intended to be used for verification through simulation, for timing analysis, for test analysis (testability analysis and fault grading), and logic synthesis.

The coding can be done in Verilog by dividing the circuit into modules. Define the modules and interconnections. The communication between modules and their environments is done using ports. The ports are of three types: input, output, inout.

Module is defined as: `module type> (list>);`

Modules need drivers to interact with ports.

It can store value.

It connects two points, but cannot store value.

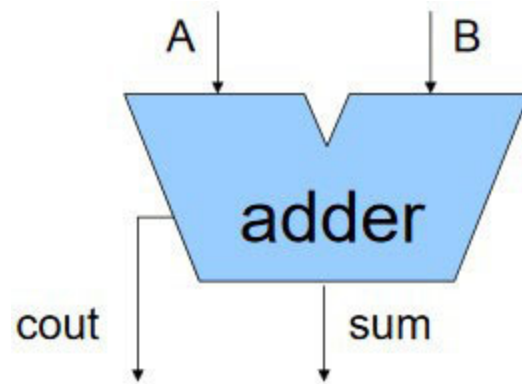


Figure 2.2: Example of adder

In fig 2.2. the module can be written in the following way:

```
module adder (A,B, sum, cout);
```

```
Input A,B;
```

```
output sum, cout;
```

```
endmodule
```

2.3. Modeling styles in Verilog HDL

A module can be designed in several ways using Verilog HDL. There are four types of abstraction levels, as per the designer specifications. That is:

Gate level

Data flow

Behavioral

Switch level

Out of these four, behavioral has the highest level of abstraction, and the switch level has the lowest level of abstraction.

2.3.1. Gate-level modeling

In gate-level modeling, the design can be implemented using logic gates, and Verilog HDL has a predefined set of logic gates. As this style is closer to physical implementation, so, gate-level modeling is also known as structural modeling. The main features of gate-level modeling are:

This modeling style is similar to the schematic drawing, where components are well connected with signals.

The module implementation is possible in terms of logic gates and interconnections between these gates.

Concurrency is possible in the gate-level modeling style.

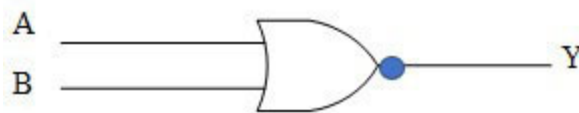


Figure 2.3: Two-input NOR gate

Verilog HDL using structural modeling/gate-level modeling

```
module xyz(A,B,Y);  
input A,B;  
output Y;  
nor N1(Y,A,B);  
endmodule
```

So, the above mentioned code is the gate-level representation of the NOR gate, comprising of output and input.

Verilog HDL for logic circuits with delays

During a simulation, sometimes, it is necessary to introduce delay. Delay is specified in terms of time units and represented by a # symbol. For example, there is a 20ns delay in the implementation of the NOR gate. So, using *figure* the Verilog HDL for NOR with delay is:

```
module xyz(A,B,Y);  
input A,B;  
output Y;  
#20 nor N1(Y,A,B);  
endmodule
```

2.3.2. Data flow modeling

The data flow model is based on logic equations and keyword assign. It is used in writing the code for Boolean expressions.

At this level, the module is designed by specifying the data flow.

Looking towards this design, one can realize how data flows between hardware registers and how the data is processed in the design.

This style is similar to logical equations. The specification is comprised of expressions made up of input signals and assigned to outputs.

In the data flow, AND operation is written as the symbol $\&$, OR operation with the symbol $|$, NOT (complement) is with symbol $\sim$.

Verilog code using data flow modeling

```
module xyz(A, B,Y);  
input A,B;  
output Y;  
assign Y=~(A|B);  
endmodule
```

In most cases, such an approach can be quite easily translated into a structure and then implemented.

Question Write the Verilog HDL for Boolean expression:

$$Y = AB + BC + AD$$

$$Z = AB +$$

Solution:

```
module my_expression(A,B,C,D,Y,Z);  
input A,B,C,D;  
output Y,Z;
```

```
assign Y=(A&B)|(B&C)|(A&D);  
assign Z=(A&B)|(~B&C)|(~A&D);  
endmodule
```

2.3.3. Behavioral modelling

Behavioral modeling is related to the algorithm or behavior of the circuit. So, it is also known as algorithmic modeling.

Behavioral modeling is the highest level of abstraction provided by Verilog HDL.

A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details.

It specifies the circuit in terms of its expected behavior.

It is the closest to a natural language description of the circuit functionality, but also the most difficult to synthesize.

Verilog code using behavioral modeling

```
module xyz(P, Q,R);  
input P,Q;  
output R;  
reg R;  
always @(P or Q)  
begin  
Y=~(P|Q);  
end  
endmodule
```

2.3.4. Switch level modeling

Switch level modeling is the lowest level of abstraction provided by Verilog. It consists of PMOS and NMOS switches.

A module can be implemented in terms of transistors, switches, storage nodes, and the interconnections between them.

In Verilog, HDL transistors are known as Switches that can either conduct or open.

Design at this level requires knowledge of switch-level implementation details.

The format for PMOS, NMOS and CMOS are:

```
nmos (out, data, ncontrol);  
pmos (out, data, pcontrol);  
cmos (out, data, ncontrol, pcontrol);
```

Verilog code using switch level modeling

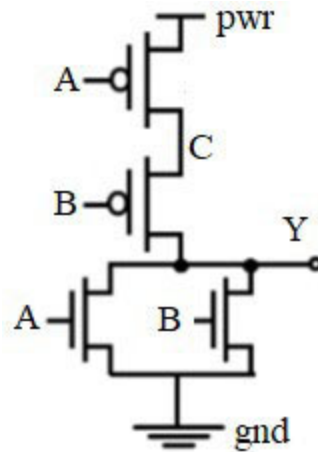


Figure 2.4: NOR gate as switches

```
module my_nor(Y,A,B);  
  output Y;  
  input A,B;  
  wire C;  
  supply1 pwr;  
  supply0 gnd ;  
  pmos (C, pwr, B);  
  pmos (Y,C,A);  
  nmos (Y, gnd, A);  
  nmos (Y, gnd, B);  
endmodule
```


Conclusion

In this chapter, we studied the different styles of modeling using Verilog Hardware Description Language. All the four modeling styles have been considered in detail using examples also.

In the next chapter, we will study the importance of all the logic gates as well as logic gate implementation using Verilog Hardware Description Language.

Questions

Explain the various types of HDL?

Differentiate between VHDL and Verilog?

What do you understand by modeling?

How many types of modeling are there for digital designing?

What is the difference between Verilog and C language?

Name any six HDL?

CHAPTER 3

Logic Gate Implementation in Verilog HDL

3.1. Logic gates

The logic gate is the building block of the digital system. One or more binary signals are inserted into the logic gate as an input, and it produces a single output. Using logic gates, various logic circuits can be designed using transistors and resistors, etc. The signals that are applied on the input and output terminal of a logic gate can be at a logic level HIGH or logic level LOW. The function of the logic gate is expressed in terms of the truth table. The truth table has all the possible combinations of input variables and the output variable.

The number of possible combinations N of binary input to a logic gate is:

$$N = 2^n$$

Where n = number of bits

For 2-bit binary number, the possible combinations are $2^2 = 4$

For 3-bit binary number, the possible combinations are $2^3 = 8$

Each variable can be at either 0 or 1, i.e., logic level LOW or logic level HIGH

Logic gates are and electronic circuit elements which perform Boolean algebraic functions. There are a total of seven logic gates, i.e., AND, NAND, OR, NOR, NOT, EX-OR, EX-NOR, out of which three are basic logic gates: AND, OR, NOT, and two are universal logic gates: NAND and NOR. These gates are called universal gates because any logic gate can be derived from these gates.

Structure

In this chapter, we will discuss the following topics:

Logic gates and its need for digital design

Working of Basic and advanced logic gates

Implementation of logic gates using Verilog HDL.

Objectives

After studying this unit, you should be able to:

Understand the design prospective of logic gates and interconnects

Design the logic gates using Verilog HDL

CMOS based implementation of logic gates

3.1.1. AND gate

Definition:

AND gate is a logic circuit whose output is 1, i.e., logic level HIGH when all the input variables are at level 1, i.e., at logic level HIGH.

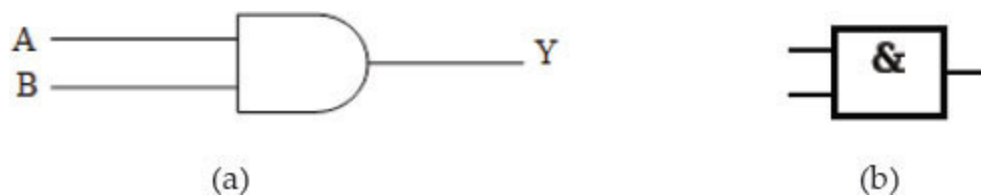


Figure 3.1: Symbol of AND Gate (a) MIL/ANSI Symbol (b) IEC symbol

The symbol of AND Gate is shown in [Figure 3.1. \(a\)](#) and In the AND function, two or more inputs are applied at the input terminal, and output is received at the receiver terminal. The output is HIGH (logic 1) only when all the inputs are high. The same is depicted in truth [table](#)

The expression for AND gate is shown as:

$$Y = A \cdot B$$

The table, which shows the correlation between inputs and outputs, is known as the truth table. The truth table of the AND gate is shown in [table](#)

Table 3.1: *Truth Table of AND Gate*

When any of the input is LOW level, i.e., 0, then output is also LOW level, i.e., level 0. The output becomes HIGH only in the case when both the inputs are at a HIGH level, i.e., level 1. The logic function of the AND gate is shown in [Figure](#)

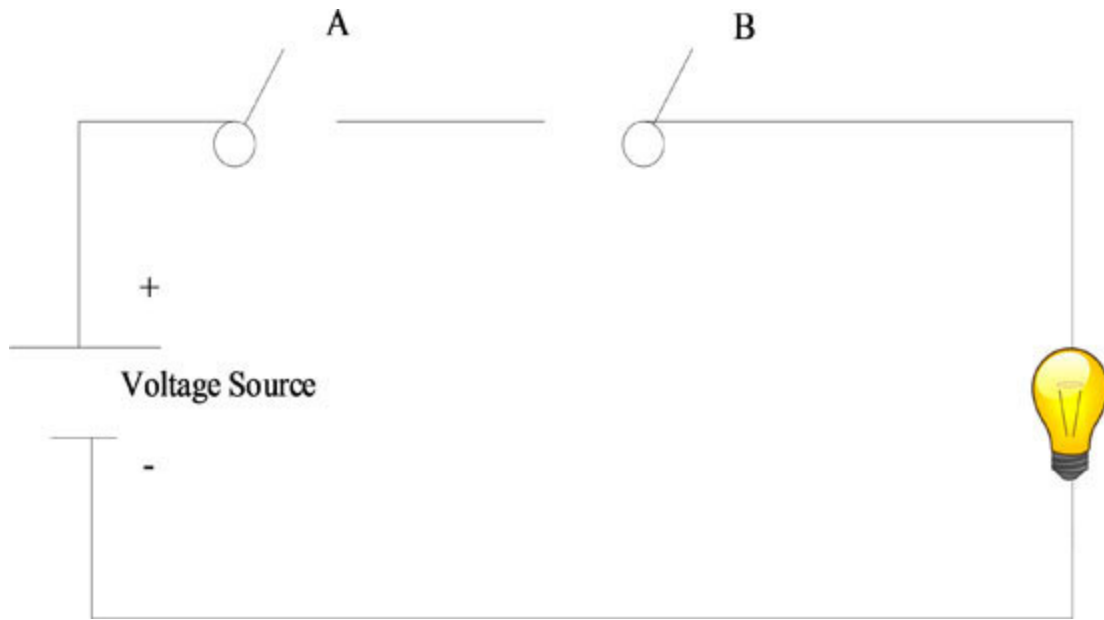


Figure 3.2: *The logic function of AND gate*

In [Figure](#) the logic function of the AND gate is discussed. Two inputs A and B, act as switches. The circuit will be completed only when both switches A and B are closed, and in that case, the bulb will start glowing. The switch is closed when input is high, i.e., logic level 1 and switch is open when input is low, i.e., logic level 0. So, for the bulb turns on switch A and switch B should be closed. In other conditions, the bulb will remain off.

3.1.1.1 Verilog program for two-input AND gate in dataflow modeling

Main module:

```
module ANDdata(a,b,y); //main module for AND gate
input a,b;
output y;
assign y=a & b; //'&' operator is been used
endmodule
```

Test bench module:

```
module ANDdataTB(); //the test bench doesn't have any port
reg p,q;
wire y;
ANDdata A1(p,q,y); //Module instantiation; here A1 is the
instance name
initial
begin
p=1'b0; q=1'b0;
#2 p=1'b0; q=1'b1;
#2 p=1'b1; q=1'b0;
#2 p=1'b1; q=1'b1;
```

```
#2 $stop;  
end  
endmodule
```

3.1.1.2 Verilog program for two-input AND gate using structural modeling

Main module:

```
module ANDgate(a,b,y); //main module for AND gate
input a,b;
output y;
and A1(y,a,b); //Primitive instantiation; here A1 is the instance
name of the primitive 'and'
endmodule //the output port comes first then the rest of the
ports
```

Test bench module:

```
module ANDgateTB; //the parenthesis is not mandatory
reg p,q;
wire y;
ANDgate A2(p,q,y); //Module instantiation; here A2 is the
instance name
initial
begin
p=1'b0; q=1'b0;
#2 p=1'b0; q=1'b1;
```

```
#2 p=1'b1; q=1'bo;
```

```
#2 p=1'b1; q=1'b1;
```

```
#2 $stop;
```

```
end
```

```
endmodule
```

3.1.1.3 Verilog program for two-input AND gate in behavioral modeling

Main module:

```
module ANDbehav(a,y,b); //main module for AND gate
output reg y; //the output port should also be declared as 'reg'
input a,b;
always @(*) //'*' represents all inputs
y=a & b; //'procedural assignment'; no assign keyword is used
endmodule //'begin' & 'end' is not used because there is a
single statement for the always block
```

Test bench module:

```
module ANDbehavTB();
wire y;
reg a,b;
ANDbehav A2(a,y,b); //Module instantiation; here A2 is the
instance name
initial
begin
a=0; b=0; //as 'a' & 'b' are 1-bit registers, assigning values
without specifying the size & type
```

#2 a=0; b=1; //will also have the same effect as “1’bo” or
“1’b1”

#2 a=1; b=0;

#2 a=1; b=1;

#2 \$stop;

end

endmodule

3.1.1.4 Verilog program for two input AND gate in switch level modeling

CMOS based circuit:

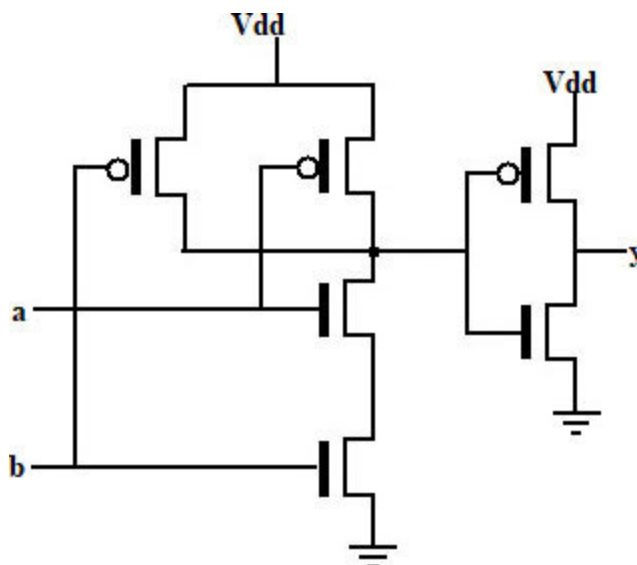


Figure 3.3: CMOS based AND gate

Main module:

```
module ANDswitch(a,y,b); //main module for AND gate
output y;
input a,b;
supply1 vdd; //for 'logic 1' connectivity
supply0 gnd; //for 'logic 0' connectivity
```



```

wire wo,w1; //for intermediate nodes
pmos p1(wo,vdd,a); // Primitive instantiation; here p1 is the
instance name
pmos p2(wo,vdd,b); // of the primitive 'pmos'

nmos n1(w1,gnd,a);
nmos n2(wo,w1,a);
pmos p3(y,vdd,wo);
nmos n3(y,gnd,wo);
endmodule

```

Test bench module:

```

module ANDswitchTB();
wire y;
reg a,b;
ANDswitch A2(a,y,b); //Module instantiation; here A2 is the
instance name
initial
begin
a=0; b=0; //as 'a' & 'b' are 1-bit registers, assigning values
without specifying the size & type
#2 a=0; b=1; //will also have the same effect as "1'b0" or
"1'b1"
#2 a=1; b=0;
#2 a=1; b=1;
#2 $stop;

```

```
end  
endmodule
```

3.1.2. OR gate

Definition:

OR gate is a type of logic gate, where output is HIGH i.e. logic 1 when at least one input is HIGH i.e. logic 1.

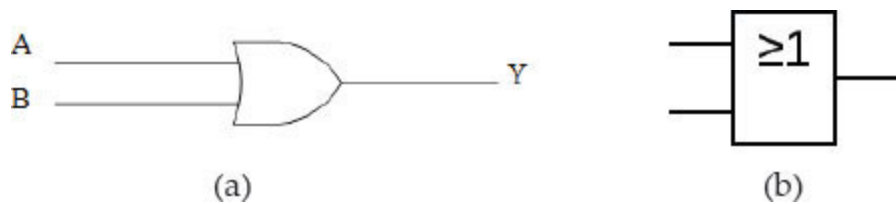


Figure 3.4: Symbol of OR Gate (a) MIL/ANSI Symbol (b) IEC symbol

The symbol of OR Gate is shown as in [Figure 3.4. \(a\)](#) and In the OR function, two or more inputs are applied at input terminal and output is received at receiver terminal. The output is HIGH (Logic 1) only when at least one input is high. The same is depicted in truth [table](#)

The expression for OR gate is shown as:

$$Y = A + B$$

The table, which shows the correlation between inputs and outputs, is known as the truth table. The truth table of the OR gate is shown in [table](#)

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Table 3.2: Truth Table of OR Gate

When any of the input is HIGH level, i.e., 1, then output is also HIGH level, i.e., level 1. The output becomes LOW only in the case when both the inputs are at a LOW level, i.e., level 0. The logic function of the OR gate is shown in [Figure](#)

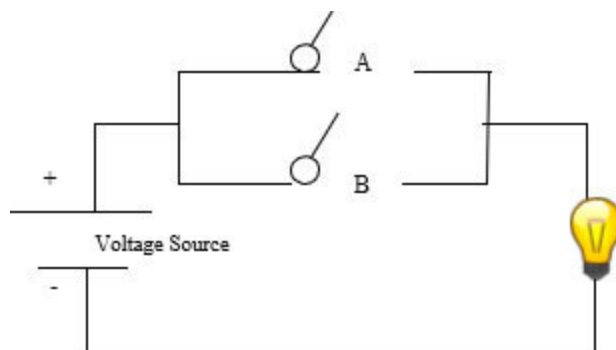


Figure 3.5: The logic function of the OR gate

In [Figure](#) the logic function of the OR gate is discussed. Two inputs A and B, act as switches. The circuit will be completed only when any of switches A and B is closed, and, in that case, the bulb will start glowing. The switch is closed when any input is high, i.e., logic level 1 and switch is open when all the input is low, i.e., logic level 0. So, for the bulb to turn on, either switch A or switch B should be closed. In other conditions, when both A and B are open, the bulb will remain off.

3.1.2.1 Verilog program for two-input OR gate in dataflow modeling

Main module:

```
module ORdata(a,b,y); //main module for OR gate
input a,b;
output y;
assign y=a | b; //'|' operator has been used
endmodule
```

Test bench module:

```
module ORdataTB(); //the test bench doesn't have any port
reg p,q;
wire y;
ORdata O1(p,q,y); //Module instantiation; here O1 is the
instance name
initial
begin
p=1'b0; q=1'b0;
#2 p=1'b0; q=1'b1;
#2 p=1'b1; q=1'b0;
#2 p=1'b1; q=1'b1;
```

```
#2 $stop;  
end  
endmodule
```

3.1.2.2 Verilog program for two-input OR gate in gate-level modeling

Main module:

```
module ORgate(a,b,y); //main module for OR gate
input a,b;
output y;
or O1(y,a,b); //Primitive instantiation; here O1 is the instance
name of the primitive 'and'
endmodule //the output port comes first then the rest of the
ports
```

Test bench module:

```
module ORgateTB; //the parenthesis is not mandatory
reg p,q;
wire y;
ORgate O2(p,q,y); //Module instantiation; here O2 is the
instance name
initial
begin
p=1'bo; q=1'bo;
#2 p=1'bo; q=1'b1;
```



```
#2 p=1'b1; q=1'bo;
```

```
#2 p=1'b1; q=1'b1;
```

```
#2 $stop;
```

```
end
```

```
endmodule
```

3.1.2.3 Verilog program for two input OR gate in behavioral modeling

Main module:

```
module ORbehav(a,y,b); //main module for OR gate
output reg y; //the output port should also be declared as 'reg'
input a,b;
always @(*) //'*' represents all inputs
begin
if (a==1) //conditional statement; if-else
y=1'b1;
else if (b) // short hand notation for (b==1)
y=1'b1;
else if (!a && !b) //short hand notation for (a==0 && b==0)
y=1'b0;
else
y=1'bx;
end
endmodule
```

Test bench module:

```
module ORbehavTB();
```

```
wire y;  
reg a,b;  
ORbehav O2(a,y,b); //Module instantiation; here O2 is the  
instance name  
initial  
begin  
a=0; b=0; //as 'a' & 'b' are 1-bit registers, assigning values  
without specifying the size & type  
#2 a=0; b=1; //will also have the same effect as "1'b0" or  
"1'b1"  
  
#2 a=1; b=0;  
#2 a=1; b=1;  
#2 $stop;  
end  
endmodule
```

3.1.2.4 Verilog program for two input OR gate in switch-level modeling

CMOS based circuit:

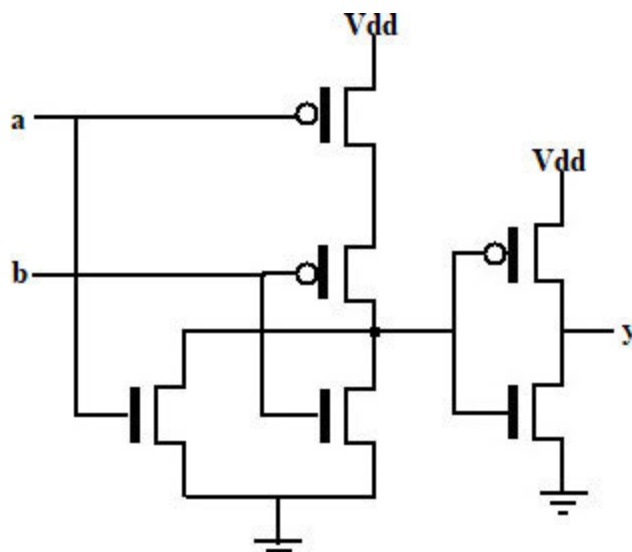


Figure 3.6: CMOS based OR gate

Main module:

```
module ORswitch(a,y,b); //main module for OR gate
output y;
input a,b;
supply1 vdd; //for 'logic 1' connectivity
supply0 gnd; //for 'logic 0' connectivity
```

```

wire w0,w1; //for intermediate nodes
pmos p1(w0,vdd,a); // Primitive instantiation; here p1 is the
instance name
pmos p2(w1,w0,b); // of the primitive 'pmos'

nmos n1(w1,gnd,a);
nmos n2(w1,gnd,a);
pmos p3(y,vdd,w1);
nmos n3(y,gnd,w1);
endmodule

```

Test bench module:

```

module ORswitchTB();
wire y;
reg a,b;
ORswitch A1(a,y,b); //Module instantiation; here A1 is the
instance name
initial
begin
a=0; b=0;
#2 a=0; b=1;
#2 a=1; b=0;
#2 a=1; b=1;
#2 $stop;
end
endmodule

```

3.1.3. NOT gate

Definition:

A logic gate is said to act as NOT gate when it changes the applied input logic level to the opposite logic level. It is also called an inverting buffer.

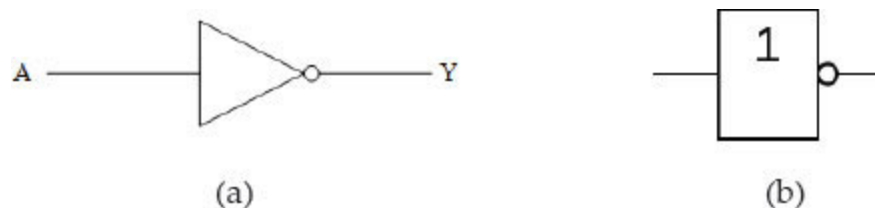


Figure 3.7: Symbol of NOT Gate (a) MIL/ANSI Symbol (b) IEC symbol

The symbol of NOT Gate is shown in [Figure 3.7.\(a\)](#) and

The NOT gate performs inversion or complement operation, where the logic level of the input variable is changed to the opposite level. For example, if 0 is applied at the input terminal, the output will provide value 1, and if 1 is applied at

the input terminal, the output will provide value 0. The same is depicted in the truth table.

The expression for NOT gate is shown as:

$$Y = \bar{A}$$

The table, which shows the correlation between inputs and outputs, is known as the truth table. The truth table of NOT gate is shown in [table](#)

Table 3.3: Truth Table of OR Gate

When the input is at a HIGH level, i.e., 1, then the output will be at a LOW level, i.e., 0 and when input is at a LOW level, i.e., 0, then the output will be at a HIGH level, i.e., 1.

3.1.3.1 Verilog program for two input NOT gate in dataflow modeling

Main module:

```
module NOTdata(b,y); //main module for NOT gate
input b;
output y;
assign y=~ b; //'~' operator has been used
endmodule
```

Test bench module:

```
module NOTdataTB(); //the test bench doesn't have any port
reg p;
wire y;
NOTdata N1(p,y); //Module instantiation; here O1 is the
instance name
initial
begin
p=1'b0;
#2 p=1'b1;
#2 $stop;
end
```


endmodule

3.1.3.2 Verilog program for two input NOT gate in gate-level modeling

Main module:

```
module NOTgate(a,y); //main module for OR gate
input a;
output y;
not n1(y,a); //Primitive instantiation; here n1 is the instance
name of the primitive 'and'
endmodule //the output port comes first then the rest of the
ports
```

Test bench module:

```
module NOTgateTB; //the parenthesis is not mandatory
reg p;
wire y;
NOTgate N1(p,y); //Module instantiation; here N1 is the
instance name
initial
begin
p=1'b0;
#2 p=1'b1;
```

```
#2 $stop;  
end  
endmodule
```

3.1.3.3 Verilog program for two input OR gate in behavioral modeling

Main module:

```
module NOTbehav(a,y); //main module for OR gate
output reg y; //the output port should also be declared as 'reg'
input a;
always @(*) //'*' represents all inputs
y=a?1'b0:1'b1; //using tertiary operator
endmodule
```

Test bench module:

```
module NOTbehavTB();
wire y;
reg a;
NOTbehav N2(a,y); //Module instantiation; here O2 is the
instance name
initial
begin
a=0; //as 'a' is 1-bit register, assigning values without specifying
the size & type
#2 a=1; //will also have the same effect as "1'b0" or "1'b1"
```

```
#2 $stop;  
end  
endmodule
```

3.1.3.4 Verilog program for two input NOT gate in switch-level modeling

CMOS based circuit:

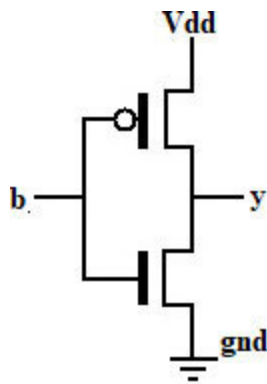


Figure 3.8: CMOS based NOT gate

Main module:

```
module NOTswitch(y,b); //main module for OR gate
output y;
input b;
supply1 vdd; //for 'logic 1' connectivity
supply0 gnd; //for 'logic 0' connectivity
pmos p3(y,vdd,b); // Primitive instantiation; here p3 is the
instance name
nmos n3(y,gnd,b);
```

```
endmodule
```

Test bench module:

```
module NOTswitchTB;
```

```
  wire y;
```

```
  reg b;
```

```
  NOTswitch N1(y,b); //Module instantiation; here N1 is the  
  instance name
```

```
  initial
```

```
  begin
```

```
    b=0;
```

```
    #2 b=1;
```

```
    #2 $stop;
```

```
  end
```

```
endmodule
```

3.2. Advanced logic gates

The AND, OR and NOT gate are the basic logic gates. Using these logic gates, any other logic function can be derived or analysed. The following gates are obtained using basic logic gates.

NAND gate

NOR gate

EX-OR gate

EX-NOR gate

3.2.1. NAND gate

Definition:

NAND gate is a logic gate whose output is LOW when all inputs are HIGH

From a combination of basic gates AND, OR, or NOT, more logic functions are derived. NAND gate is one of the advanced versions of a primary gate which is derived from AND and NOT gate. This gate is called a universal gate because any primary gate can be driven from this gate.

As the NAND gate is a combination of AND and NOT gate, it implies an AND function with complemented output. The symbol and its equivalence are shown in [Figure](#)

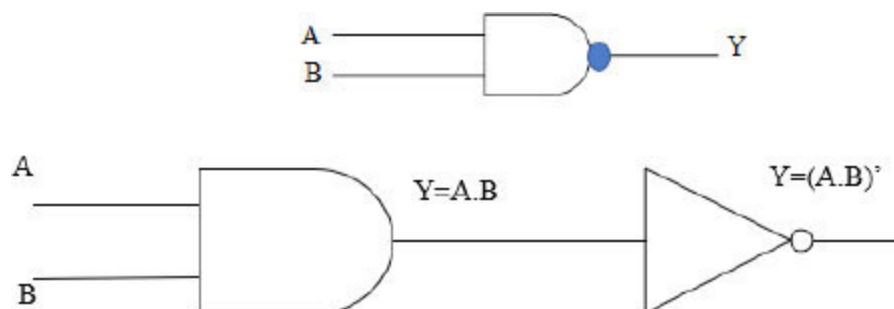


Figure 3.9: NAND gate (a) symbol (b) equivalent

In the NAND gate logic function, two or more inputs are applied at the input terminal, and output is received at the receiver terminal. The output is HIGH (logic 1) when any of the input variables is low. The same is depicted in truth [table](#)

The expression for NAND gate is shown as:

$$Y = (A.B)^{-}$$

The table, which shows the correlation between inputs and outputs, is known as the truth table. The truth table of the NAND gate is shown in [table](#)

Table 3.4: Truth Table of NAND Gate

When any of the input is LOW level, i.e., level 0, then the output will be at a HIGH level, i.e., level 1. The output becomes LOW only in the case when both the inputs are at a HIGH level, i.e., level 1.

Example Elaborate the operation of NAND gate as negative OR gate.

The operation of the NAND gate and OR gate is shown in [table](#)

Table 3.5: *The truth table of OR and NAND gate*

From the above table, we can easily deduce that the operation of the NAND gate and OR gate are in a complementary manner, or we can say that the aspect of the NAND gate operation is negative OR.

Further, it can be shown with the help of logic gate, as depicted in [Figure](#)

As, we can see bubbles are present at input terminals, so input variables become $\bar{A}$ and $\bar{B}$. The OR of these two variables are:

$\bar{A} + \bar{B} = A\bar{B}$, which is the NAND gate.

Example 3.2: Prove that NAND is a universal logic gate.

A universal logic gate is that property of a logic gate, which enables it to create primary gate AND, OR, and NOT from it.

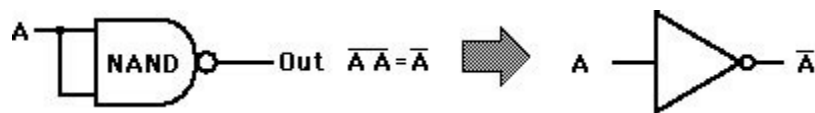


Figure 3.10: (a) NAND gate as NOT gate

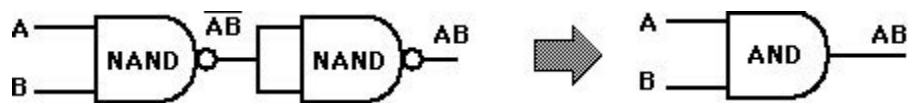


Figure 3.10: (b) NAND gate as AND gate

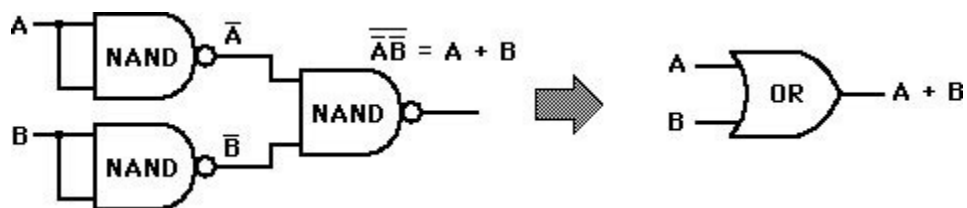


Figure 3.10: (c) NAND gate as OR gate

Hence proved, NAND is a universal logic gate, because using only NAND gate, other basic gates are derived successfully.

3.2.1.1 Verilog_program for two input NAND_gate in dataflow modeling

Main module:

```
module NANDdata(a,b,y); //main module for NAND gate
input a,b;
output y;
assign y=~(a & b);
endmodule
```

Test bench module:

```
module NANDdataTB(); //the test bench doesn't have any port
reg p,q;
wire y;
NANDdata ND1(p,q,y); //Module instantiation; here ND1 is the
instance name
initial
begin
p=1'bo; q=1'bo;
#2 p=1'bo; q=1'b1;
#2 p=1'b1; q=1'bo;
#2 p=1'b1; q=1'b1;
```

```
#2 $stop;  
end  
endmodule
```

3.2.1.2 Verilog program for two input NAND gate in gate-level modeling

Main module:

```
module NANDgate(B,b,y); //main module for NAND gate
input B,b; //Verilog is a case sensitive language; 'b' & 'B' will
be treated as two different
output y; // identifiers
nand ND1(y,B,b); //Primitive instantiation; here ND1 is the
instance name of the primitive
endmodule //‘nand’; the output port comes first then the rest
of the ports
```

Test bench module:

```
module NANDgateTB; //the parenthesis is not mandatory
reg p,q;
wire y;
NANDgate ND2(p,q,y); //Module instantiation; here ND2 is the
instance name
initial
begin
p=1'bo; q=1'bo;
```

```
#2 p=1'bo; q=1'b1;  
#2 p=1'b1; q=1'bo;  
#2 p=1'b1; q=1'b1;  
#2 $stop;  
end  
endmodule
```


3.2.1.3 Verilog program for two input NAND gate in behavioral modeling

Main module:

```
module NANDbehav(a,y,b); //main module for NAND gate
output reg y; //the output port should also be declared as 'reg'
input a,b;
always @(*) //'*' represents all inputs
begin
if (a==0) //conditional statement; if-else
y=1'b1;
else if (!b) // short hand notation for (b==0)
y=1'b1;
else if (a && b) //short hand notation for (a==1 && b==1)
y=1'b0;
else
y=1'bx;
end
endmodule
```

Test bench module:

```
module NANDbehavTB();
```

```
wire y;  
reg a,b;  
NANDbehav ND1 (a,y,b); //Module instantiation; here ND1 is  
the instance name  
initial  
begin  
a=0; b=0; //as 'a' & 'b' are 1-bit registers, assigning values  
without specifying the size & type  
#2 a=0; b=1; //will also have the same effect as "1'b0" or  
"1'b1"  
  
#2 a=1; b=0;  
#2 a=1; b=1;  
#2 $stop;  
end  
endmodule
```

3.2.1.4 Verilog program for two input NAND gate in switch-level modeling

CMOS based circuit:

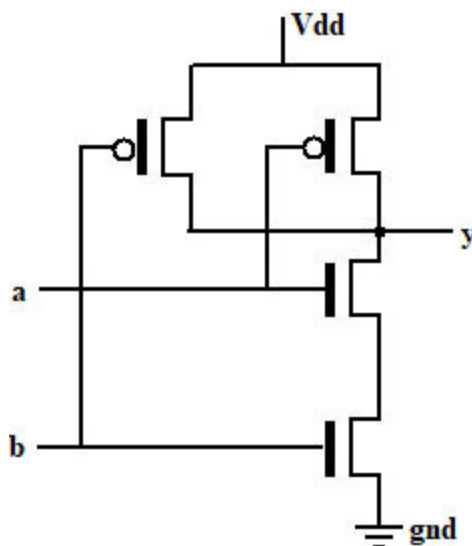


Figure 3.11: CMOS based NAND gate

Main module:

```
module NANDswitch(a,y,b); //main module for NAND gate
output y;
input a,b;
supply1 vdd; //for 'logic 1' connectivity
supply0 gnd; //for 'logic 0' connectivity
```

```
wire wo;
pmos p1(y,vdd,a); // Primitive instantiation; here p1 is the
instance name
pmos p2(y,vdd,b); // of the primitive 'pmos'

nmos n1(wo,gnd,a);
nmos n2(y,wo,a);
endmodule
```

Test bench module:

```
module NANDswitchTB();
wire y;
reg a,b;
NANDswitch ND1(a,y,b); //Module instantiation; here ND1 is
the instance name
initial
begin
a=0; b=0;
#2 a=0; b=1;
#2 a=1; b=0;
#2 a=1; b=1;
#2 $stop;
end
endmodule
```

3.2.2 NOR gate

Definition:

NOR gate is a logic gate where output is HIGH only when both inputs are LOW.

From combination of basic gates AND, OR or NOT, more logic functions are derived. NOR gate is one of the advanced versions of basic gate which is derived from OR and NOT gate. This gate is called universal gate because any basic gate can be driven from this gate.

As NOR gate is combination of OR and NOT gate, it implies an OR function with complemented output. The symbol and its equivalence are shown as in [Figure](#)

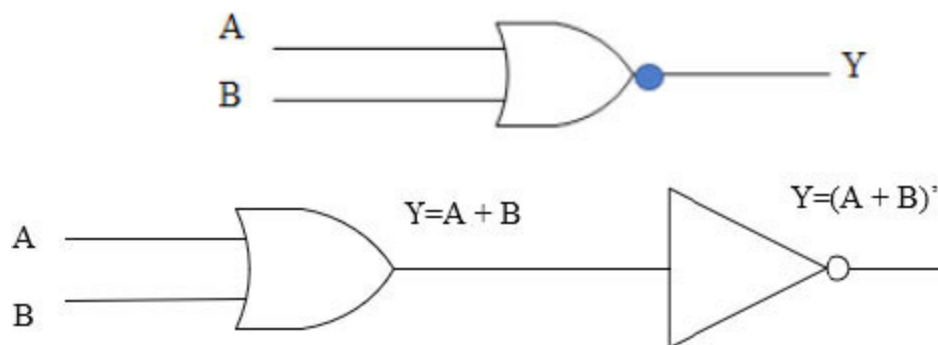


Figure 3.12: NOR gate (a) symbol (b) equivalent

In the NOR gate logic function, two or more inputs are applied at the input terminal, and output is received at the receiver terminal. The output is HIGH (logic 1) when all input variables are low. The same is depicted in truth [table](#). The expression for NOR gate is shown as:

$$Y = A+B^{-}$$

The table, which shows the correlation between inputs and outputs, is known as the truth table. The truth table of the NOR gate is shown in [table](#)

Table 3.6: Truth Table of NOR gate

When any of the input is HIGH level, i.e., level 1, then the output will be at a LOW level, i.e., level 0. The output becomes LOW only in the case when both the inputs are at a LOW level, i.e., level 0.

Example 3.3: Elaborate the operation of NOR gate as negative AND gate.

The operation of the NOR gate and AND gate is shown in [table](#)

Table 3.7: *The truth table of OR and NAND gate*

From the above table, we can easily deduce that the operation of the NOR gate and AND gate is in a complementary manner, or we can say that the aspect of the NOR gate operation is negative AND.

Further, it can be shown with the help of logic gate and Boolean algebra, as in [Figure](#)

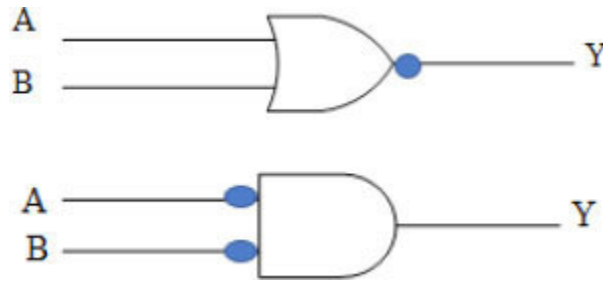


Figure 3.13: NOR as Negative AND gate

As, we can see bubbles are present at input terminals, so input variables become $\bar{A}$ and $\bar{B}$. The OR of these two variables are:

$\bar{A} + \bar{B} = A \cdot B$, which is the negative OR gate.

Example 3.4: Prove that NOR is a Universal logic gate

A universal logic gate is that property of a logic gate, which enables it to create primary gate AND, OR, and NOT from it.



Figure 3.14: (a) NOR gate as NOT gate

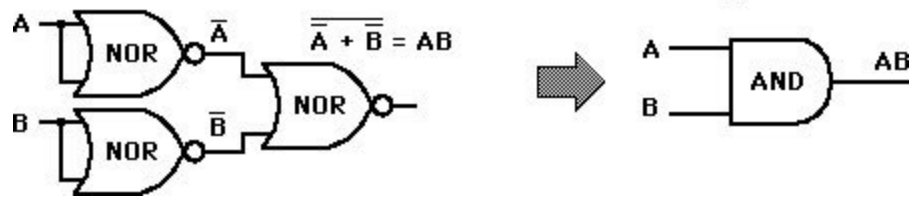


Figure 3.14: (b) NOR gate as AND gate

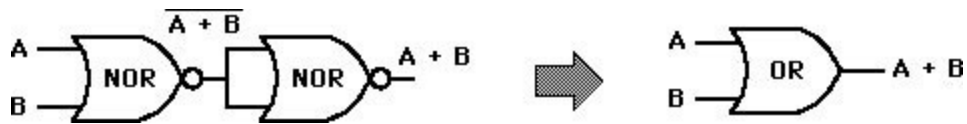


Figure 3.14: (c) NOR gate as OR gate

Hence proved, NOR is a universal logic gate, because using only NOR gate, other basic gates are derived successfully.

3.2.2.1 Verilog_program for two input NOR gate in dataflow modeling

Main module:

```
module NORdata(a,b,y); //main module for NOR gate
input a,b;
output y;
assign y=~(a | b);
endmodule
```

Test bench module:

```
module NORdataTB(); //the test bench doesn't have any port
reg p,q;
wire y;
NANDdata NR1(p,q,y); //Module instantiation; here NR1 is the
instance name
initial
begin
p=1'bo; q=1'bo;
#2 p=1'bo; q=1'b1;
#2 p=1'b1; q=1'bo;
#2 p=1'b1; q=1'b1;
```

```
#2 $stop;  
end  
endmodule
```

3.2.2.2 Verilog program for two input NOR gate in gate-level modeling

Main module:

```
module NORgate(B,b,y); //main module for NOR gate
input B,b; //Verilog is a case sensitive language; 'b' & 'B' will
be treated as two different
output y; // identifiers
nor NR1(y,B,b); //Primitive instantiation; here NR1 is the
instance name of the primitive
endmodule //‘nand’; the output port comes first then the rest
of the ports
```

Test bench module:

```
module NORgateTB; //the parenthesis is not mandatory
reg p,q;
wire y;
NORgate ND2(p,q,y); //Module instantiation; here ND2 is the
instance name
initial
begin
p=1'bo; q=1'bo;
```

```
#2 p=1'bo; q=1'b1;  
#2 p=1'b1; q=1'bo;  
#2 p=1'b1; q=1'b1;  
#2 $stop;  
end  
endmodule
```

3.2.1.3 Verilog program for two input NAND gate in behavioral modeling

Main module:

```
module NORbehav(a,y,b); //main module for NOR gate
output reg y; //the output port should also be declared as 'reg'
input a,b;
always @(*) //'*' represents all inputs
begin
case({a,b})
2'b00; y=1;
2'b01; y=0;
2'b10; y=0;
2'b11; y=0;
endcase
end
endmodule
```

Test bench module:

```
module NORbehavTB();
wire y;
reg a,b;
```

```
NORbehav ND1 (a,y,b); //Module instantiation; here ND1 is the
instance name
initial
begin
a=0; b=0; //as 'a' & 'b' are 1-bit registers, assigning values
without specifying the size & type
#2 a=0; b=1; //will also have the same effect as "1'b0" or
"1'b1"
#2 a=1; b=0;
#2 a=1; b=1;

#2 $stop;
end
endmodule
```

3.2.1.4 Verilog program for two input NOR gate in switch-level modeling

CMOS Based circuit:

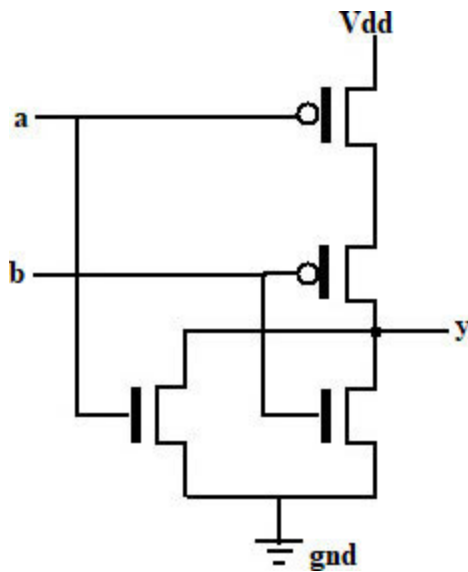


Figure 3.15: CMOS based NOR gate

Main module:

```
module NORswitch(a,y,b); //main module for NOR gate
output y;
input a,b;
supply1 vdd; //for 'logic 1' connectivity
supply0 gnd; //for 'logic 0' connectivity
```



```
wire wo;
pmos p1(wo,vdd,a); // Primitive instantiation; here p1 is the
instance name
pmos p2(y,wo,b); // of the primitive 'pmos'

nmos n1(y,gnd,a);
nmos n2(y,gnd,b);
endmodule
```

Test bench module:

```
module NORswitchTB();
wire y;
reg a,b;
NORswitch NR1(a,y,b); //Module instantiation; here NR1 is the
instance name
initial
begin
a=0; b=0;
#2 a=0; b=1;
#2 a=1; b=0;
#2 a=1; b=1;
#2 $stop;
end
endmodule
```

3.2.3. EX-OR or XOR gate

Definition:

EX-OR is a logic gate that where output is HIGH when only one input is HIGH.

The symbol of EX-OR (XOR) gate is shown as in [Figure](#). In EX-OR gate, two or more inputs are applied and output is received at receiver terminal.



Figure 3.16: Symbol of EX-OR gate

The truth table depicts the relation between input variables and output variables. The Boolean expression of EX-OR gate is shown as:

$$Y = A \oplus \bar{B}$$

Where $A \oplus \bar{B} = A\bar{B} + \bar{A}B$

The table, which shows the correlation between inputs and outputs, is known as the truth table. The truth table of the EX-OR gate is shown in [table](#)

Table 3.8: Truth table of EX-OR gate

When any of the input is HIGH level, i.e., level 1, then the output will be at a HIGH level, i.e., level 1. In the EX-OR gate, the output is HIGH when an odd number of inputs is HIGH when the even number of input variables is HIGH; it will produce output LOW. This property of the EX-OR gate differs from the OR gate.

3.2.3.1 Verilog program for two input XOR gate in dataflow modeling

Main module:

```
module XORdata(a,b,y); //main module for XOR gate
input a,b;
output y;
assign y=a ^ b); // '^' is the XOR operator
endmodule
```

Test bench module:

```
module XORdataTB(); //the test bench doesn't have any port
reg p,q;
wire y;
XORdata XR1(p,q,y); //Module instantiation; here XR1 is the
instance name
initial
begin
p=1'b0; q=1'b0;
#2 p=1'b0; q=1'b1;
#2 p=1'b1; q=1'b0;
#2 p=1'b1; q=1'b1;
```

```
#2 $stop;  
end  
endmodule
```

3.2.3.2 Verilog program for two input XOR gate in gate-level modeling

Main module:

```
module XORgate(a,b,y); //main module for XOR gate
input a,b;
output y;
xor XR1(y,a,b); //Primitive instantiation; here XR1 is the instance
name of the primitive
endmodule //‘xor’; the output port comes first then the rest of
the ports
```

Test bench module:

```
module XORgateTB; //the parenthesis is not mandatory
reg p,q;
wire y;
XORgate XR2(p,q,y); //Module instantiation; here XR2 is the
instance name
initial
begin
p=1'bo; q=1'bo;
#2 p=1'bo; q=1'b1;
```

```
#2 p=1'b1; q=1'bo;
```

```
#2 p=1'b1; q=1'b1;
```

```
#2 $stop;
```

```
end
```

```
endmodule
```

3.2.3.3 Verilog program for two input XOR gate in behavioral modeling

Main module:

```
module XORbehav(a,y,b); //main module for XOR gate
output reg y; //the output port should also be declared as 'reg'
input a,b;
always @(*) //'*' represents all inputs
begin
case({a,b})
2'b00; y=0;
2'b01; y=1;
2'b10; y=1;
2'b11; y=0;
endcase
end
endmodule
```

Test bench module:

```
module XORbehavTB();
wire y;
reg a,b;
```



```
XORbehav XR1 (a,y,b); //Module instantiation; here XR1 is the
instance name
initial
begin
a=0; b=0; //as 'a' & 'b' are 1-bit registers, assigning values
without specifying the size & type
#2 a=0; b=1; //will also have the same effect as "1'b0" or
"1'b1"
#2 a=1; b=0;
#2 a=1; b=1;

#2 $stop;
end
endmodule
```

3.2.3.4 Verilog program for two input XOR gate in switch-level modeling

CMOS based circuit:

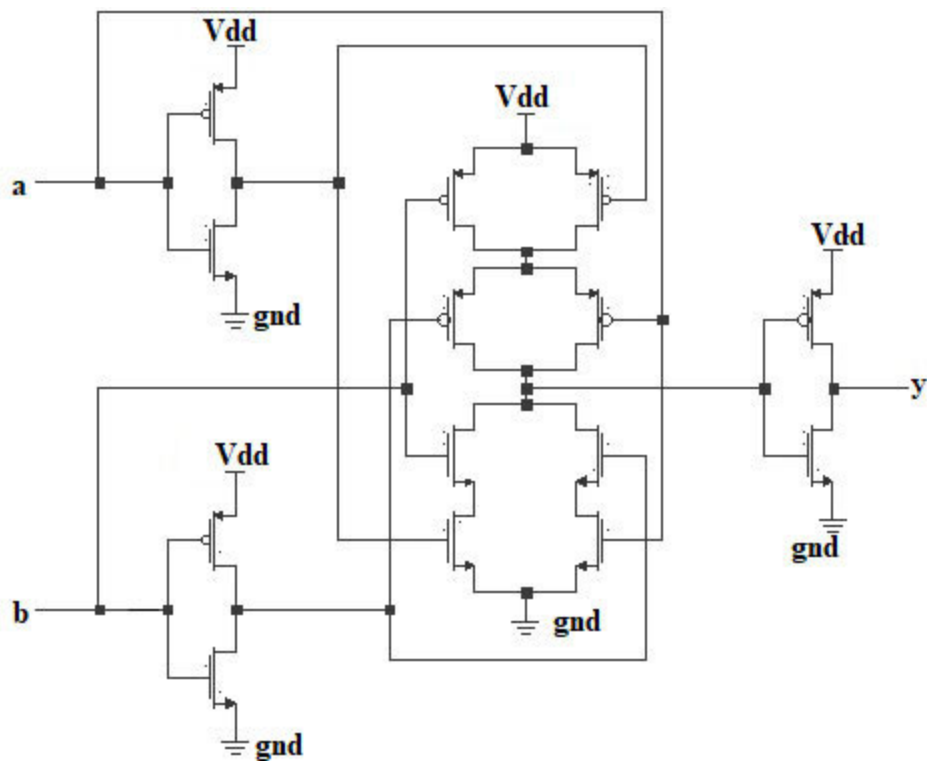


Figure 3.17: CMOS based XOR gate

Main module:

```
module XORswitch(a,y,b); //main module for XOR gate
```

```

output y;
input a,b;
supply1 vdd; //for 'logic 1' connectivity
supply0 gnd; //for 'logic 0' connectivity
wire w0,w1,w2,w4,w5;
pmos p1(w0,vdd,a); // Primitive instantiation; here p1 is the
instance name
nmos n1(w0,gnd,a);
pmos p2(w1,vdd,b);

nmos n2(w1,gnd,b);
pmos p3(w2,vdd,w0);
pmos p4(w2,vdd,w1);
pmos p5(y,w2,a);
pmos p6(y,w2,b);
nmos n3(w4,gnd,b);
nmos n4(w5,gnd,w1);
nmos n5(y,w4,a);
nmos n6(y,w5,w0);
endmodule

```

Test bench module:

```

module XORswitchTB();
wire y;
reg a,b;

```

```
XORswitch XR1(a,y,b); //Module instantiation; here XR1 is the  
instance name  
initial  
begin  
a=0; b=0;  
#2 a=0; b=1;  
#2 a=1; b=0;  
#2 a=1; b=1;  
#2 $stop;  
end  
endmodule
```

3.2.4. EX-NOR or XNOR gate

Definition:

EX-NOR is a logic gate that was output is HIGH when both input variables are either HIGH or LOW.

The Symbol of EX-NOR (XNOR) gate is shown as in [Figure](#) In the EX-NOR gate, two or more inputs are applied, and output is received at the receiver terminal.



Figure 3.18: Symbol of EX-NOR gate

The truth table depicts the relation between input variables and output variables. The Boolean expression of EX-NOR gate is shown as:

$$Y = A \oplus \overline{B}$$

Here, $A \oplus \overline{B} = A\overline{B} + A\overline{\overline{B}} = \overline{A}.\overline{B} + A.B$

$Y = A \oplus \overline{B}$ can also be shown as with symbol $\odot$, i.e., $A \odot B$.

The table, which shows the correlation between inputs and outputs, is known as the truth table. The truth table of the EX-NOR gate is shown in [table](#)

Table 3.9: Truth Table of EX-NOR Gate

When any of the input is HIGH level, i.e., level 1, then the output will be at the LOW level, i.e., level 0. In the EX-NOR gate, the output is HIGH when all the input variables are of the same level, i.e., HIGH or LOW.

The EX-NOR gate is also called a coincidence gate. The applications of the EX-NOR gate are:

It can be used as a 1-bit magnitude comparator. It produces HIGH output when both inputs are similar.

It can be used as a parity checker. Its output is LOW when a number of 1's is odd, and its output is HIGH when a number of 1's is even.

3.2.4.1 Verilog program for two input XNOR gate in dataflow modeling

Main module:

```
module XNORdata(a,b,y); //main module for XNOR gate
input a,b;
output y;
assign y=a ~^ b); // either of '~^' or '^~' is the XNOR
operator
endmodule
```

Test bench module:

```
module XNORdataTB(); //the test bench doesn't have any port
reg p,q;
wire y;
XNORdata XR1(p,q,y); //Module instantiation; here XR1 is the
instance name
initial
begin
p=1'b0; q=1'b0;
#2 p=1'b0; q=1'b1;
#2 p=1'b1; q=1'b0;
```



```
#2 p=1'b1; q=1'b1;  
#2 $stop;  
end  
endmodule
```

3.2.4.2 Verilog program for two input XNOR gate in gate-level modeling

Main module:

```
module XNORgate(a,b,y); //main module for XNOR gate
input a,b;
output y;
xnor XR1(y,a,b); //Primitive instantiation; here XR1 is the
instance name of the primitive
endmodule //‘xnor’; the output port comes first then the rest
of the ports
```

Test bench module:

```
module XNORgateTB; //the parenthesis is not mandatory
reg p,q;
wire y;
XNORgate XR2(p,q,y); //Module instantiation; here XR2 is the
instance name
initial
begin
p=1'bo; q=1'bo;
#2 p=1'bo; q=1'b1;
```

```
#2 p=1'b1; q=1'bo;
```

```
#2 p=1'b1; q=1'b1;
```

```
#2 $stop;
```

```
end
```

```
endmodule
```

3.2.4.3 Verilog program for two input XNOR gate in behavioral modeling

Main module:

```
module XNORbehav(a,y,b); //main module for XNOR gate
output reg y; //the output port should also be declared as 'reg'
input a,b;
always @(*) //'*' represents all inputs
begin
case({a,b})
2'b00; y=1;
2'b01; y=0;
2'b10; y=0;
2'b11; y=1;
endcase
end
endmodule
```

Test bench module:

```
module XNORbehavTB();
wire y;
reg a,b;
```

```
XNORbehav XR1 (a,y,b); //Module instantiation; here XR1 is the
instance name
initial
begin
a=0; b=0; //as 'a' & 'b' are 1-bit registers, assigning values
without specifying the size & type
#2 a=0; b=1; //will also have the same effect as "1'b0" or
"1'b1"
#2 a=1; b=0;
#2 a=1; b=1;

#2 $stop;
end
endmodule
```

3.2.4.4 Verilog program for two input XNOR gate in switch-level modeling

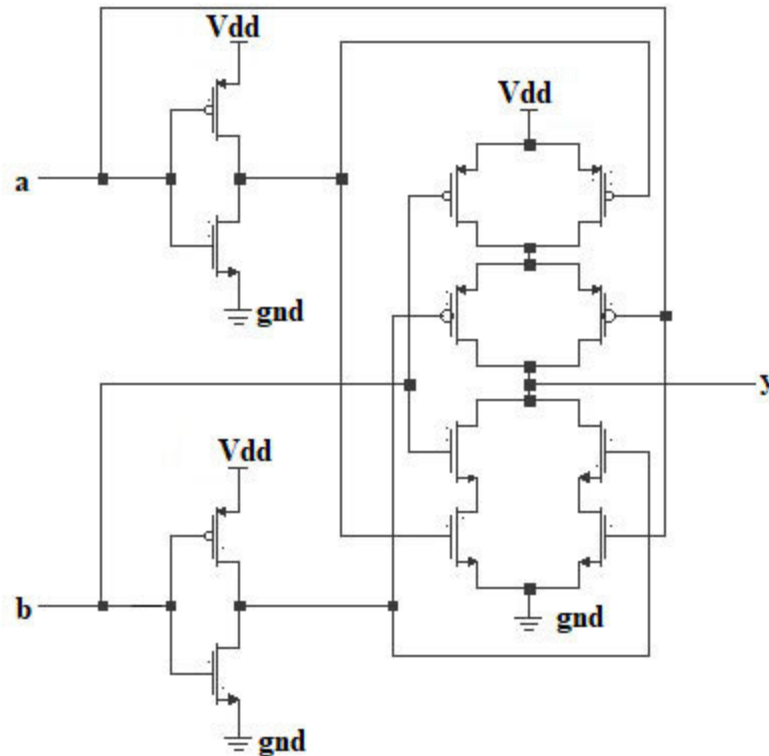


Figure 3.19: CMOS based XNOR gate

Main module:

```
module XNORswitch(a,y,b); //main module for XNOR gate
output y;
input a,b;
supply1 vdd; //for 'logic 1' connectivity
```

```

supplyo gnd; //for 'logic o' connectivity
wire w0,w1,w2,w4,w5;
pmos p1(w0,vdd,a); // Primitive instantiation; here p1 is the
instance name
nmos n1(w0,gnd,a);

pmos p2(w1,vdd,b);
nmos n2(w1,gnd,b);
pmos p3(w2,vdd,w0);
pmos p4(w2,vdd,b);
pmos p5(y,w2,a);
pmos p6(y,w2,w1);
nmos n3(w4,gnd,b);
nmos n4(w5,gnd,w1);
nmos n5(y,w4,w0);
nmos n6(y,w5,a);
endmodule

```

Test bench module:

```

module XNORswitchTB();
wire y;
reg a,b;
XNORswitch XR1(a,y,b); //Module instantiation; here XR1 is the
instance name
initial
begin

```

```
a=0; b=0;  
#2 a=0; b=1;  
#2 a=1; b=0;  
#2 a=1; b=1;  
#2 $stop;  
end  
endmodule
```


Conclusion

In this chapter, we studied the logic gates and their implementation using Verilog HDL. The CMOS based implementation for logic gates, using switch level modeling is also described in this chapter.

In the next chapter, we will study the implementation of adder and subtractor using Verilog HDL.

Questions

Implement the Boolean equation $Y=A'BC+D'$

- (a) Using gate-level modeling
- (b) Using data flow modeling
- (c) Using behavioral modeling
- (d) Using switch level modeling

Design the following equation using CMOS.

- (a) $Y=(AB+C)'$
- (b) $Y=(A+BC)'$
- (c) $Y=AB+C$
- (d) $Y=A+BC$

State the consensus theorem and design using switch level modeling.

Minimize the following function using the Karnaugh map and design it using gate-level modeling.

$$F = \sum m(1, 3, 5, 6, 7)$$

Give the IC numbers of all the logic gates.

How VHDL is different than Verilog?

CHAPTER 4

Adder-Subtractor Implementation Using Verilog HDL

4.1. Introduction

Different logic circuits can be designed using various logic gates in different combinations. The digital logic circuits are mainly classified into two types:

Combinational logic circuits

Sequential logic circuits

The combinational logic circuit is that form of digital circuit, in which output is dependent only on the present state. The logic gates are the basic building block of any digital circuits. The combinational term signifies the combination of two or more logic gates to derive an output function. For example, adder, subtractor, etc. are the example of combinational logic circuits.

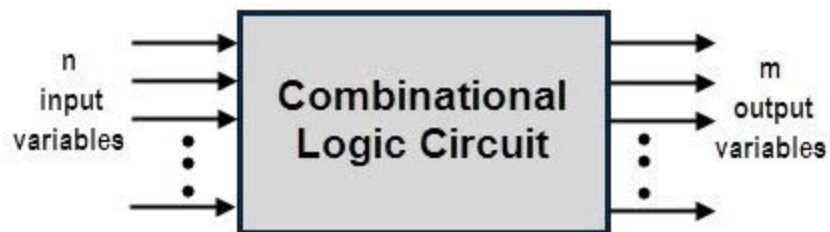


Figure 4.1: Combinational logic circuit

The [figure 4.1](#) shows the combinational logic circuits, where n -input variables generate the m -output variables, depending on the combination of logic gates.

On the other hand, the sequential logic circuits are the type of digital circuits, where output is dependent, not only the present state but also on the previous state. Memory is required in sequential logic circuits. Examples of sequential logic circuits are counters, shift registers, etc.

Structure

In this chapter, we will discuss the following topics:

Design strategy of combinational logic circuits

Half Adder, Full Adder and its implementation in Verilog HDL

Half Subtractor, Full Subtractor and its implementation in Verilog HDL

Objectives

After studying this unit, you should be able to:

Design the combinational logic circuits using k-map, boolean function, and logic gates.

Implementation of adder and subtractors using Verilog HDL.

Representation of full adder and full subtractor using sub modules.

4.2. The design strategy of combinational logic circuits

The designing of combinational logic circuits involves the following steps:

Understand the nature of the problem.

Identify the number of input and output variables and analyze the relationship in between input and output variables.

Formulate the truth table that explores all the output states in tabular form for each possible combination of the input variable.

Express the input and output in terms of a Boolean expression.

Using k-map or Boolean algebra, calculate the minimized Boolean function.

Design the minimized Boolean function using logic gates.

4.3. Adders

The adder is a digital logic circuit that implements the addition of numbers. In combinational logic circuits, an adder performs the addition of two binary bits. Depending on the number of bits, the adders are further classified into two types as half adder and full adder.

4.3.1. Half Adder

The half adder operates addition on two binary bits, augend, and addend. It produces output in the form of sum and carries.



Figure 4.2: Block diagram of half adder

As per steps, discussed in section 4.2., the number of inputs and outputs are identified and labeled it. There are two inputs, A and B producing two outputs Sum and carry. As the number of inputs is two, it suggests four possible combinations. The half adder performs binary addition. The block diagram of half adder is shown in [figure](#)

Table 4.1: The truth table of Half adder

[Table 4.1](#) shows the input-output relationship of half adder. A minimized Boolean function can be obtained using the Karnaugh map. The two-variable k-map is used to explore the Boolean function.

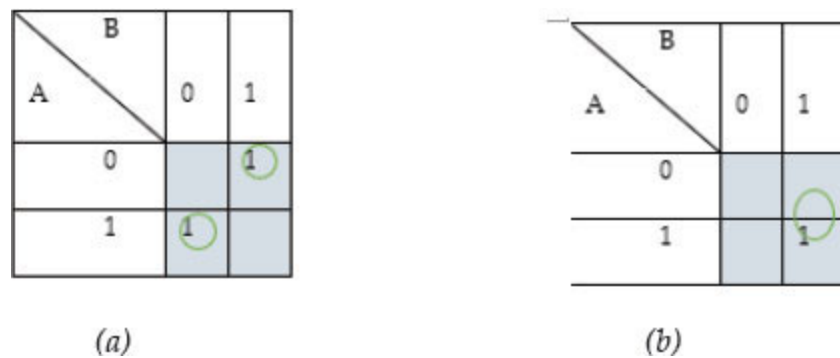


Figure 4.3: K-map for (a) Sum and (b) Carry

Using k-map, as in [figure](#) we get,

$$Sum = A\bar{B} + \bar{A}B \quad (4.1)$$

$$Carry = AB \quad (4.2.)$$

The logical representation of Sum and carry using a combination of EX-OR and AND gate is shown in [figure](#)

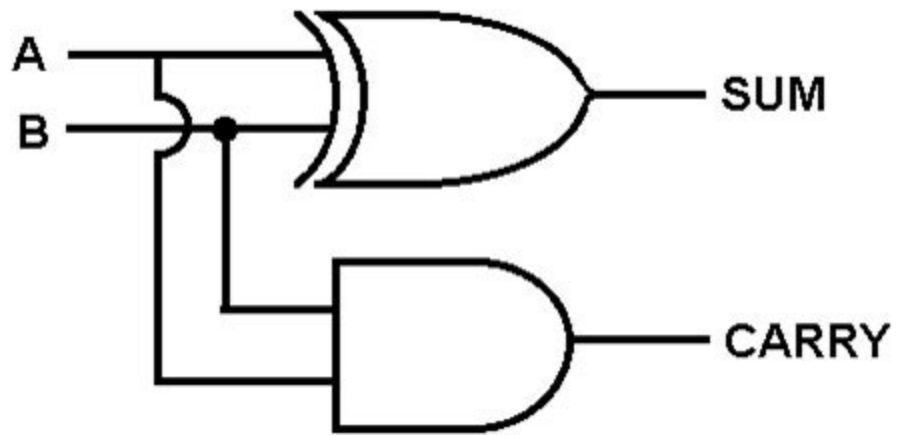


Figure 4.4: Half Adder using logic gates

4.3.2. Full adder

The full adder performs the addition of three binary bits to accommodate the carry generated by previous or low order bits. It consists of three inputs A, B, C in, and two outputs sum, carry.



Figure 4.5: Block diagram of Full Adder

The A and B are the significant binary bits, and Cin is the carry from low order bit. As the number of inputs is three, so the possible combinations are eight. The truth table is formulated, which shows the output sum and carry concerning the all possible combinations of inputs. [Table 4.2](#) shows the truth table of a full adder.

adder.
adder.

adder.
adder.
adder.
adder.
adder.
adder.
adder.
adder.

adder.

Table 4.2: *The truth table of full adder*

A minimized Boolean function can be obtained using the Karnaugh map. The three-variable k-map is used to explore the Boolean function.

BCin A		00	01	11	10
		0	1	1	1
0		1	1	1	1
1		1	1	1	1

Figure 4.6: (a) *K-map for sum*

$$Sum = ABCin\bar{+} \bar{A}\bar{B}Cin + \bar{A}BCin\bar{+} A\bar{B}Cin \quad (4.3)$$

		BCin			
		00	01	11	10
A	0			1	
	1		1	1	1

Figure 4.6: (b) K-map for carry

$$Carry = AB + ACin + BCin \quad (4.4)$$

The expression of the sum is further simplified using Boolean algebra laws and properties of logic gates.

$$\begin{aligned}
 Sum &= \overline{ABCin} + \overline{A}\bar{B}Cin + \bar{A}B\overline{Cin} + A\bar{B}\overline{Cin} \\
 &= Cin(\overline{AB} + \overline{A}\bar{B}) + \overline{Cin}(\overline{A}\bar{B} + A\bar{B}) \\
 &= Cin(\overline{A \text{ XOR } B}) + \overline{Cin}(A \text{ XOR } B) \\
 &= Cin(\overline{A \oplus B}) + \overline{Cin}(A \oplus B) \\
 &= Cin \oplus (A \oplus B) \quad (A \oplus B = \overline{A}B + A\bar{B}) \\
 &= Cin \oplus A \oplus B \\
 \text{So, } Sum &= Cin \oplus A \oplus B \quad (4.5)
 \end{aligned}$$

The expression of carry is further simplified using Boolean algebra laws and properties of logic gates.

$$\text{Carry} = AB + ACin + BCin$$

$$= AB(Cin + \overline{Cin}) + ACin(B + \overline{B}) + BCin(A + \overline{A})$$

$$= ABCin + AB\overline{Cin} + ABCin + A\overline{B}Cin + ABCin + \overline{A}BCin$$

$$= ABCin + AB\overline{Cin} + A\overline{B}Cin + \overline{A}BCin \quad (A + A = A)$$

$$= AB(Cin + \overline{Cin}) + Cin(A \oplus B) \quad (A + \overline{A} = 1)$$

$$= AB + Cin(A \oplus B)$$

$$\text{So, Carry} = AB + Cin(A \oplus B) \quad (4.6)$$

The simplified Boolean expression can be represented using logic gates as:

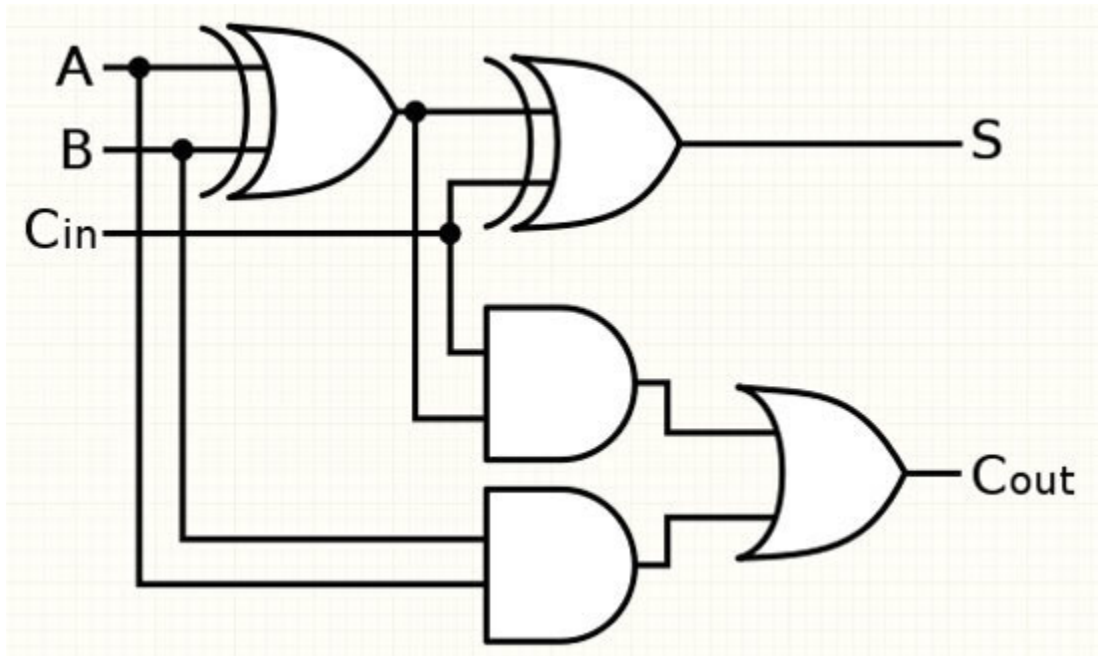


Figure 4.7: Full Adder using logic gates

Example 4.1: Represent full adder using half adder

The full adder can be represented as the combination of two half adders as [figure](#)

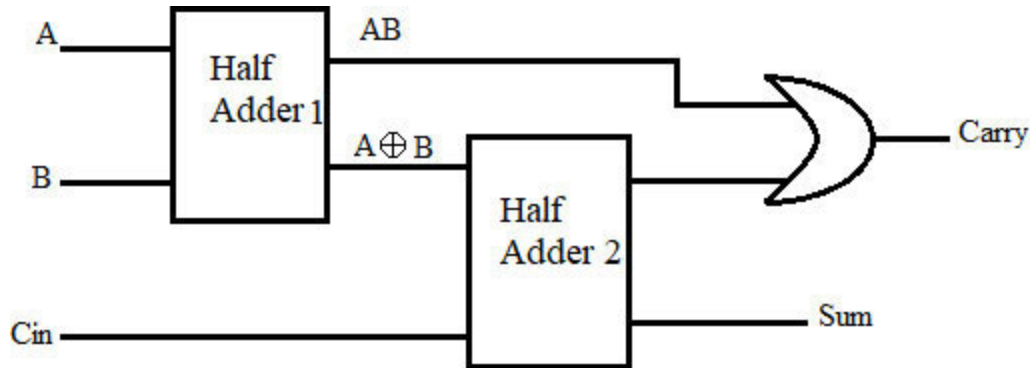


Figure 4.8: Full adder using half adder

4.3.3 Verilog program for half adder in dataflow modelling

Main module:

```
module Half(a,b,s,c);  
input a,b;  
output s,c;  
assign s=a ^ b;  
assign c=a & b;  
endmodule
```

Test bench module:

```
module HalfTB; //the test bench doesn't have any port  
reg p,q;  
wire sum,carry;  
Half H1(p,q,sum,carry); //Module instantiation; here H1 is the  
instance name  
initial  
begin  
p=1'bo; q=1'bo;  
#2 p=1'bo; q=1'b1;  
#2 p=1'b1; q=1'bo;  
#2 p=1'b1; q=1'b1;  
#2 $stop;  
end
```

endmodule

4.3.4 Verilog program for full adder using half adder

Main module:

```
module Full(a,b,cin,s,c);
input a,b;
output s,c;
wire so,co,c1;
Half H2(a,b,so,co); //Half adder designed in earlier section, is
instantiated
Half H3(so,cin,s,c1);
or o1(c,co,c1); //or' primitive is used for generating the carry
endmodule
```

Test bench module:

```
module FullTB(); //the test bench doesn't have any port
reg a,b,cin;
wire s,c;
Full f1(a,b,cin,s,c);
initial
begin
```

a=1'bo; b=1'bo;cin=1'bo; //it is not mandatory to provide all combinations

#2 a=1'bo; b=1'b1; cin=1'bo;

#2 a=1'b1; b=1'bo; cin=1'bo;

#2 a=1'b1; b=1'b1; cin=1'b1;

#2 \$stop;

end

endmodule

4.4. Subtractor

The subtractor is a digital logic circuits which implements subtraction of numbers. In combinational logic circuits, a subtractor performs the subtraction of two binary bits. Depending on the number of bits, the subtractors are further classified in two types as half subtractor and full subtractor.

4.4.1. Half subtractor

The half subtractor performs the operation of subtraction on two binary bits, minuend and subtrahend. It produces output in form of difference and borrow.

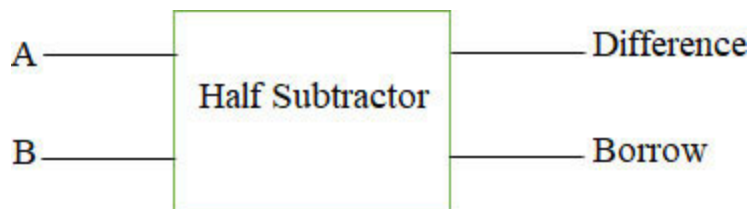


Figure 4.9: Block diagram of half subtractor

As per steps, discussed in section 4.2., the number of inputs and outputs are identified and labeled it. There are two inputs, A and B producing two outputs Difference and Borrow. As the number of inputs is two, it suggests four possible combinations. The half subtractor performs binary subtraction. [Figure](#) shows the block diagram of half subtractor.

subtractor.
subtractor.

subtractor.
subtractor.
subtractor.
subtractor.

Table 4.3: *The truth table of Half Subtraction*

[Table 4.3](#) shows the input-output relationship of half subtractor. A minimized Boolean function can be obtained using the Karnaugh map. The two-variable k-map is used to explore the Boolean function.

A \ B	0	1
0	0	1
1	0	0

(a)

A \ B	0	1
0	0	1
1	1	0

(b)

Figure 4.10: K-map for (a) Difference and (b) Borrow

Using K-map, as in [figure](#) we get,

$$\text{Difference} = \bar{A}B \quad (4.7)$$

$$\text{Borrow} = A\bar{B} + \bar{A}B \quad (4.8)$$

The logical representation of Difference and Borrow using combination of EX-OR and AND gate, is shown as in [figure](#)

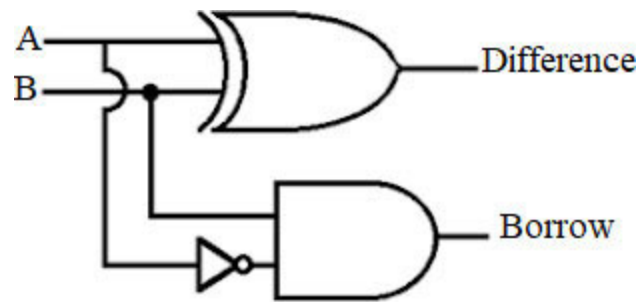


Figure 4.11: Half subtractor using logic gates

4.4.2. Full subtractor

The full subtractor performs the subtraction of three binary bits to accommodate the borrow bit, generated by low order bits. It consists of three inputs A, B, Bin, and two outputs difference, borrow.

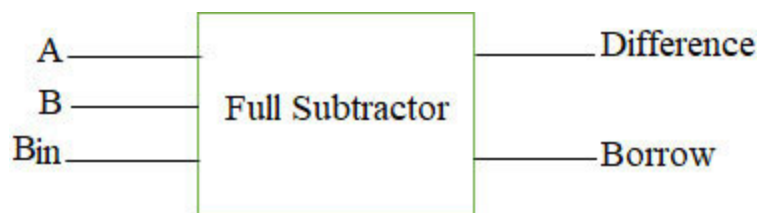


Figure 4.12: Block diagram of full subtractor

The A and B are the significant binary bits, and Bin is the borrow from the low order bit. As the number of inputs is three, so the possible combinations are eight. The truth table is formulated, which shows the output difference and borrow concerning the all possible combinations of inputs. The block diagram of the full subtractor is shown in [figure Table 4.4](#). shows the truth table of full subtractor.

subtractor.

subtractor.
subtractor.
subtractor.
subtractor.
subtractor.
subtractor.
subtractor.
subtractor.

subtractor.

Table 4.4: *The truth table of full subtractor*

A minimized Boolean function can be obtained using the Karnaugh map. The three-variable k-map is used to explore the Boolean function, as in [figure](#)

BBin		00	01	11	10
A					
0			1		1
1		1		1	

Figure 4.13: (a) K-map for Difference

$$\text{Difference} = AB\bar{B}in + \bar{A}\bar{B}Bin + \bar{A}B\bar{B}in + A\bar{B}\bar{B}in \quad (4.9)$$

BBin		00	01	11	10
A					
0			1	1	1
1				1	

Figure 4.13: (b) K-map for Borrow

$$\text{Borrow} = \bar{A}B + BBin + \bar{A}Bin \quad (4.10)$$

The expression of difference is further simplified using Boolean algebra laws and properties of logic gates.

$$\begin{aligned}
 \text{Difference} &= AB\bar{B}in + \bar{A}\bar{B}Bin + \bar{A}B\bar{B}in + A\bar{B}\bar{B}in \\
 &= Bin(AB + \bar{A}\bar{B}) + \bar{B}in(\bar{A}B + AB) \\
 &= Bin(A \text{ XNOR } B) + \bar{B}in(A \text{ XOR } B) \\
 &= Bin(\overline{A \text{ XOR } B}) + \bar{B}in(A \text{ XOR } B) \\
 &= Bin \text{ XOR } (A \text{ XOR } B) \quad (A \oplus B = \bar{A}B + A\bar{B}) \\
 &= Bin \oplus A \oplus B
 \end{aligned}$$

$$\text{So, Difference} = Bin \oplus A \oplus B \quad (4.5)$$

The expression of borrow is further simplified using Boolean algebra laws and properties of logic gates.

$$\text{Borrow} = \bar{A}B + BBin + \bar{A}Bin$$

The simplified Boolean expression can be represented using logic gates, as in [figure](#)

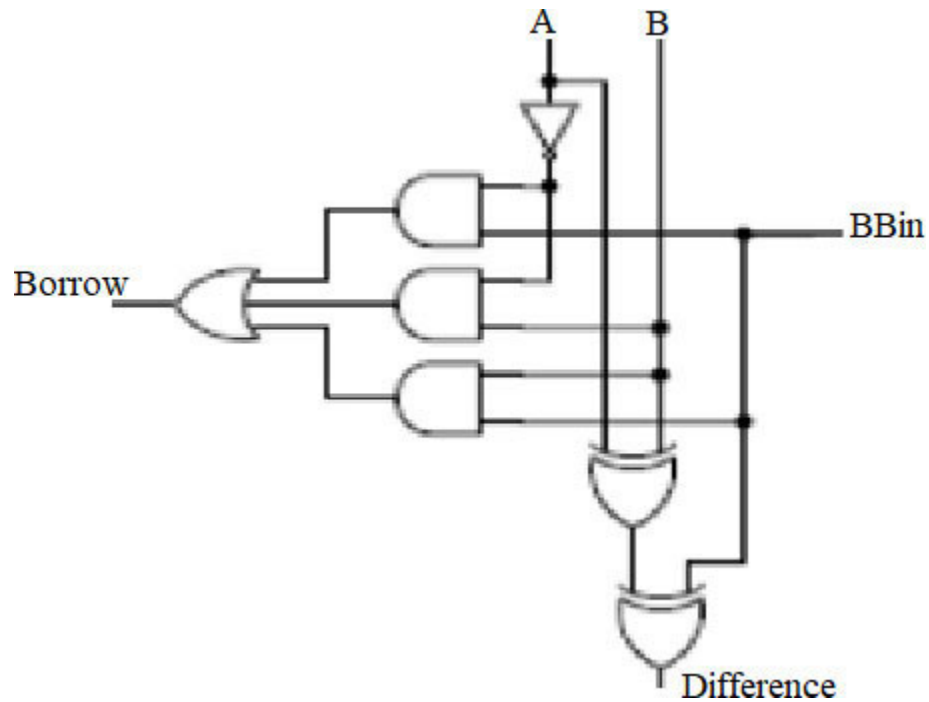


Figure 4.14: Full subtractor using logic gates

Example 4.2: Design full subtractor using half subtractor.

The full subtractor can be designed using half subtractor as [figure](#)

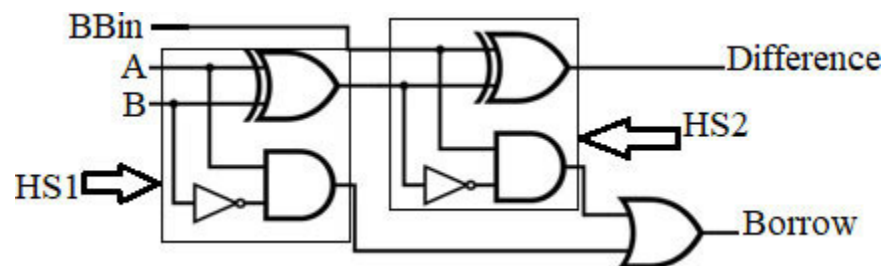


Figure 4.15: *Full subtractor using half subtractor*

4.4.3 Verilog program for half subtractor in behavioral modelling

The behavioral modelling is related with the algorithm or behavior of the circuit. So, it is known as algorithmic modelling. This is the highest level of abstraction provided by Verilog HDL. A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. The behavioral code of half subtractor is shown below.

Main module:

```
module HalfSub(a,b,d,bo);  
input a,b;  
output reg d,bo;  
always @(*)  
begin  
d=a ^ b;  
bo=(~a) & b;  
end  
endmodule
```

Test bench module:


```
module HalfSubTB; //the test bench doesn't have any port
reg p,q;
wire diff,borrow;
HalfSub H1(p,q, diff,borrow); //Module instantiation; here H1 is
the instance name
initial
begin
p=1'bo; q=1'bo;
#2 p=1'bo; q=1'b1;
#2 p=1'b1; q=1'bo;

#2 p=1'b1; q=1'b1;
#2 $stop;
end
endmodule
```

4.4.4 Verilog program for full subtractor using half subtractor

Main module:

```
module FullSub(a,b,bin,d,bo);  
input a,b,bin;  
output d,bo;  
wire do,boo,bo1;  
HalfSub H2(a,b,do,boo); //Half subtractor designed in earlier  
section, is instantiated  
HalfSub H3(do,bin,d,bo1);  
or o1(bo,boo,bo1); //or' primitive is used for generating the  
borrow  
endmodule
```

Test bench module:

```
module FullSubTB(); //the test bench doesn't have any port  
reg a,b,bin;  
wire d,bo;  
FullSub f1(a,b,bin,d,bo);  
initial  
begin
```

a=1'bo; b=1'bo;bin=1'bo; //it is not mandatory to provide all combinations

#2 a=1'bo; b=1'b1; bin=1'bo;

#2 a=1'b1; b=1'bo; bin=1'bo;

#2 a=1'b1; b=1'b1; bin=1'b1;

#2 \$stop;

end

endmodule

4.4.5 Verilog program for full subtractor using half subtractor

Main module:

```
module FullSub(a,b,bin,d,bo);  
input a,b,bin;  
output d,bo;  
wire do,boo,bo1;  
HalfSub H2(a,b,do,boo); //Half subtractor designed in earlier  
section, is instantiated  
HalfSub H3(do,bin,d,bo1);  
or o1(bo,boo,bo1); //or' primitive is used for generating the  
borrow  
endmodule
```

Test bench module:

```
module FullSubTB(); //the test bench doesn't have any port  
reg a,b,bin;  
wire d,bo;  
FullSub f1(a,b,bin,d,bo);  
initial  
begin
```

a=1'bo; b=1'bo;bin=1'bo; //it is not mandatory to provide all combinations

#2 a=1'bo; b=1'b1; bin=1'bo;

#2 a=1'b1; b=1'bo; bin=1'bo;

#2 a=1'b1; b=1'b1; bin=1'b1;

#2 \$stop;

end

endmodule

4.5. Verilog code for adder cum subtractor

The adder-cum-subtractor or also known as adder/subtractor has a dual operation, i.e., works as adder as well as a subtractor. [Figure 4.16](#) shows the logic diagram of n-bit parallel adder-cum-subtractor, realized using n full adders.

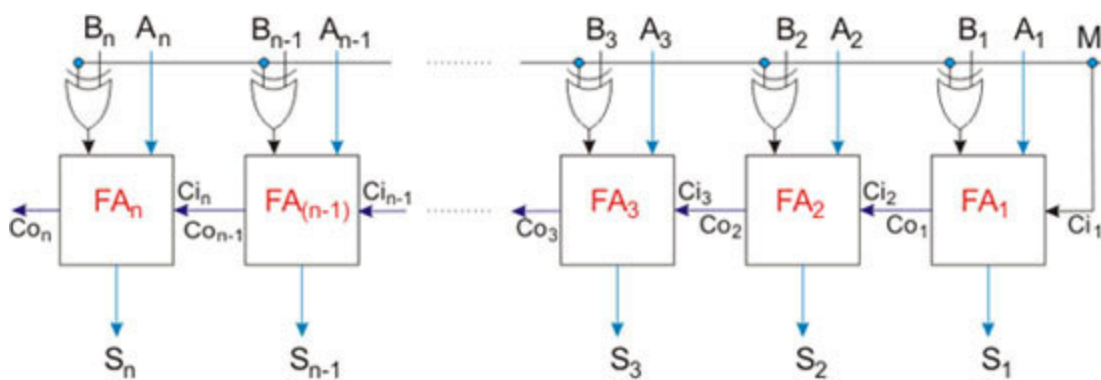


Figure 4.16: N-bit parallel adder/subtractor circuit realized using n full adders

```
module FA (A,B,Cin,Sum,Carry);  
input A,B,Cin;  
output Sum,Carry;  
assign Sum=A^B^Cin;  
assign Carry=(A&B)|(Cin&(A^B));  
endmodule
```

```

module AdderCumSub(A,B,M,S,Co);
//The circuit performs the operation in 2's complement format
input [3:0] A,B;
input M; //if M=0 the circuit performs addition; if M=1 the
circuit performs subtraction
output [3:0]S;
output Co;
wire [2:0] Cout;
wire [3:0] Bxor;

xor x0(Bxor[0],B[0],M);
xor x1(Bxor[1],B[1],M);
xor x2(Bxor[2],B[2],M);
xor x3(Bxor[3],B[3],M);
FA fo(A[0],Bxor[0],M,S[0],Cout[0]);
FA f1(A[1],Bxor[1],Cout[0],S[1],Cout[1]);
FA f2(A[2],Bxor[2],Cout[1],S[2],Cout[2]);
FA f3(A[3],Bxor[3],Cout[2],S[3],Co);
Endmodule

```

Conclusion

In this chapter, we studied the design strategies of various combinational logic circuits and its implementation using Verilog HDL. The various types of adder-subtractor have been analysed along with its HDL implementation.

In the next chapter, we will study the implementation of multiplexer and de-multiplexer using Verilog HDL.

Questions

Signify the importance of combinational logic circuits.

Design 1-bit full subtractor using structural modelling.

Design half adder and half subtractor using behavioral modelling.

Design full subtractor using half subtractor using Verilog HDL

CHAPTER 5

Multiplexer/Demultiplexer Implementation in Verilog HDL

5.1. Introduction to multiplexer

The multiplexer is a combinational logic circuit that is designed to choose one input out of various inputs. Here, 'various' refers to 2^n , n is several data inputs. The function of the multiplexer is controlled by a select line or control signal. The shortened form of the multiplexer is MUX or MPX. The multiplexer is also known as a data selector, as it 'selects' input data to be produced on the output terminal.

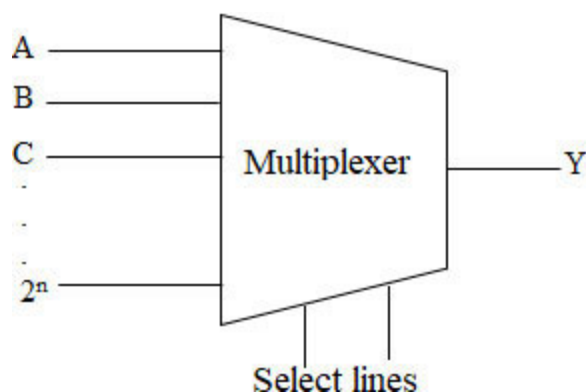


Figure 5.1: Multiplexer

[Figure 5.1](#) shows the block diagram of the multiplexer, where several inputs are 2^n and select lines are 'n.'

Structure

In this chapter, we will discuss the following topics:

Introduction to multiplexer and de-multiplexer

Implementation of multiplexer and de-multiplexer using Verilog
HDL

Objectives

After studying this unit, you should be able to:

Understand the significance of multiplexer and de-multiplexer.

Design of 2:1, 4:1, 8:1 multiplexer and 1:2, 1:4 and 1:8 de-multiplexer using Verilog HDL.

5.2. The 4:1 Multiplexer

The 4 to 1 multiplexer, the 4 represents the number of inputs, and 1 represents several outputs. The number of select lines will be 2 ($2^2=4$; $n=2$). The fig.7.2. shows the configuration of the 4:1 multiplexer.

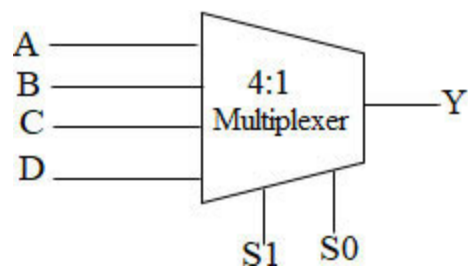


Figure 5.2: 4:1: multiplexer

[Table 5.1](#) shows the truth table of 4 to 1 multiplexer. From the truth table, the logical expression can be derived as:

as:
as:
as:
as:

as:
as:

Table 5.1: *The truth table of 4:1 multiplexer*

The Select line 'So and S1' will decide which input will go to the output terminal.

The output $Y=A$, when $S_1=0$ and $S_0=0$

The output $Y=B$, when $S_1=0$ and $S_0=1$

The output $Y=C$, when $S_1=1$ and $S_0=0$

The output $Y=D$, when $S_1=1$ and $S_0=1$

Therefore, $Y=A.\overline{S_1}.\overline{S_0} + B.\overline{S_1}.S_0 + C.S_1.\overline{S_0} + D.S_1.S_0$

The logic representation of 4:1 multiplexer using logic gates is shown as in [*figure*](#)

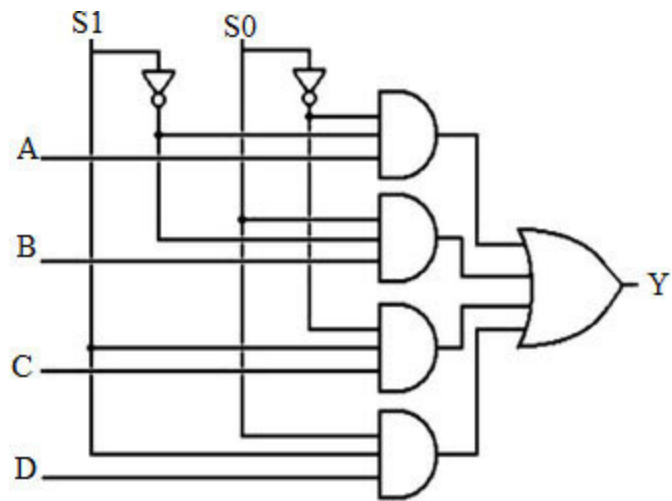


Figure 5.3: Logic diagram of 4:1 multiplexer

5.3. IC 74153 pin diagram

IC 74153 contains two 4:1 multiplexer. Each multiplexer has strobe input G₁ and G₂ at pin 1 and pin 15, respectively. For valid output, the strobe input has to be at logic 0. The select lines are at pin 2 and pin 14. Two sets of inputs can be given to IC74153, i.e., pin 10-13 or pin 3-6. The output can be taken at pin 9 or pin 7, as per the input set.

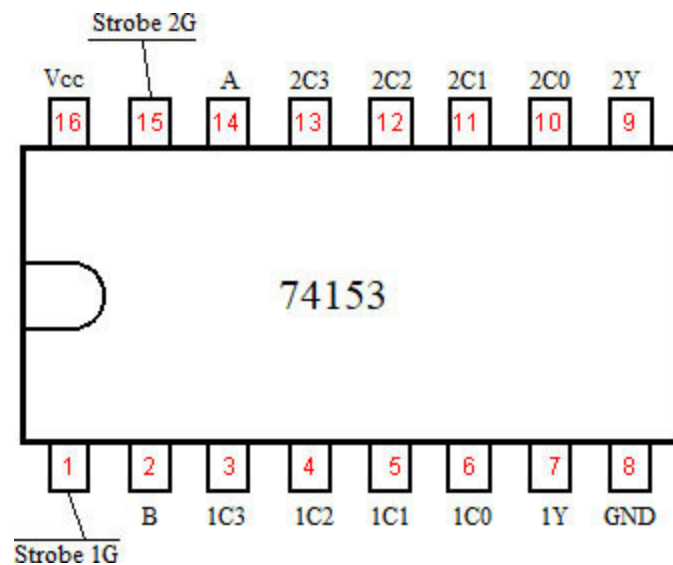


Figure 5.4: Pin diagram of IC74153

The truth table of two 4:1 multiplexer using IC74153 is shown as in [table](#)

Table 5.2: 4:1: multiplexer using IC74153

5.4. Verilog_program for 2:1 mux design

The multiplexer is implemented using data flow modelling, gate level modelling and behavioral modelling.

5.4.1. Multiplexer implementation using dataflow modeling

The data flow model is based on logic equations and keyword assign. It is used in writing the code for Boolean expressions. At this level, the module is designed by specifying the data flow.

```
module MUX_data(y,a,b,s);  
input a,b,s;  
output y;  
assign y=(a&(~s))|(b&s);  
endmodule
```

5.4.2. Multiplexer implementation using gate-level modelling

In gate level modeling, the design can be implemented using logic gates and Verilog HDL has a predefined set of logic gates. As this style is closer to physical implementation, so, gate level modeling is also known as structural modeling.

```
module MUX_gate(y,a,b,s);  
input a,b,s;  
output y;  
wire w0,w1,w2;  
not n1(w0,s);  
and a1(w1,a,w0);  
and a2(w2,b,s);  
or o1(y,w1,w2);  
endmodule
```

5.4.3. Multiplexer implementation using behavioral modeling

Behavioral modeling is related to the algorithm or behavior of the circuit. So, it is also known as algorithmic modeling. This is the highest level of abstraction provided by Verilog HDL.

```
module MUX_behav(y,a,b,s);  
input a,b,s;  
output reg y;  
always@(*)  
begin  
if (!s)  
y=a;  
else  
y=b;  
endmodule
```

5.5. Verilog_program for 4:1 mux design

The multiplexer is implemented using data flow modelling, gate level modelling and behavioral modelling.

5.5.1. Multiplexer implementation using dataflow modelling

```
module MUX41_data(y,a,b,c,d,so,s1);  
input a,b,c,d,so,s1;  
output y;  
assign y=(a&(~so)&(~s1))|(b&(~so)&s1)| (c&so&(~s1))|  
(b&so&s1);  
endmodule
```


5.5.2. Multiplexer implementation using structural modelling

```
module MUX4_1_struct(z,i,sel); //considering the input & the
select line as vector
input [3:0]i;
input [1:0]sel;
output y;
wire [1:0]w;
MUX_data m1(w[0],i[0],i[1],sel[0]); //using "MUX_data" module
MUX_data m2(w[1],i[2],i[3],sel[0]);
MUX_data m3(y,w[0],w[1],sel[1]);
endmodule
```

5.6. Verilog_program for 8:1 mux design

The multiplexer is implemented using data flow modelling, gate level modelling and behavioral modelling.

5.6.1. Multiplexer implementation using structural modelling

```
module MUX8_1_struct(out,x,select); //considering the input & the
select line as vector
input [7:0]x;
input [2:0]select;
output out;
wire [2:0]w;
MUX4_1_struct m1(w[0],x[3:0],select[1:0]); //using "MUX4_1_struct"
modules
MUX4_1_struct m2(w[1],x[7:4],select[1:0]);
MUX_data m3(out,w[0],w[1],select[2]); // using "MUX_data"
modules
endmodule
```

5.7. Introduction to demultiplexer

The demultiplexer is a combinational logic circuit where one input signal is transmitting over various outputs. It receives data information on a single channel and distributes over several output lines. So, demultiplexer is a type of combinational logic circuit where number of inputs is single and number of outputs are 2^n . The demultiplexer is also known as data distributor. It is also known as serial to parallel converter.

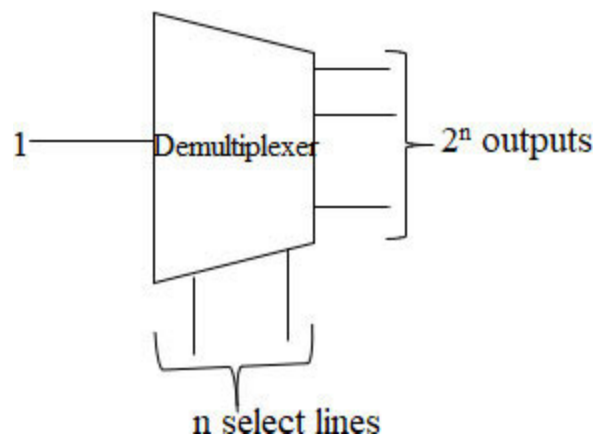


Figure 5.5: Demultiplexer

Figure 5.5 shows the block diagram of the demultiplexer, where several inputs are one, and outputs are

5.7. The 1:2 demultiplexer

A 1 to 2 demultiplexer has a single input and 2 output lines (A and B). The number of select lines is 1 select line=1). [Table 5.3](#) shows the truth table of 1 to 2 demultiplexer.

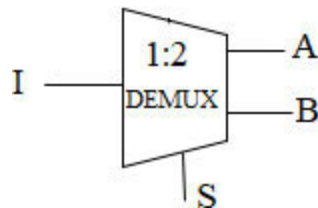


Figure 5.6: 1 to 2 demultiplexer

The truth [table 5.3](#) depicts the functionality of 1 to 2 demultiplexer. The data input I is connected to output A when the select line S is 0 and data input, I is connected to output B when the select line is 1.

1.
1.
1.
1.

1.
1.

Table 5.3: The truth table of 1 to 2 demultiplexer

The expression for outputs can be written as:

$$B = IS$$

From these expressions, a 1 to 2 demultiplexer can be implemented using logic gates, as per [Figure](#)

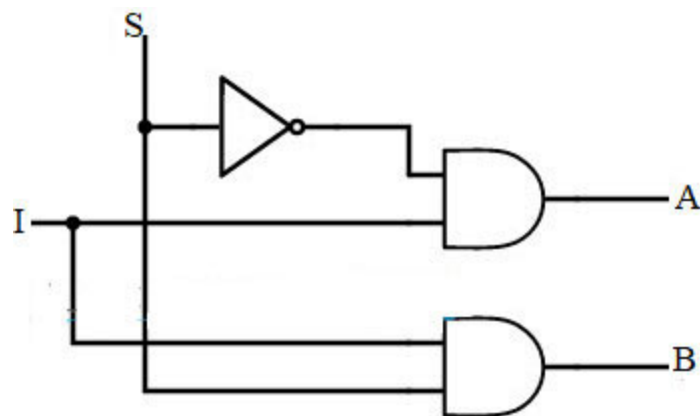


Figure 5.7: Logic diagram of 1 to 2 demultiplexer

5.8. The 1:4 demultiplexer

A 1 to 4 demultiplexer has a single input and 4 output lines (A, B, C, and D). The number of select lines is 2 select line=2). [Table 5.4](#) shows the truth table of 1 to 4 demultiplexers.

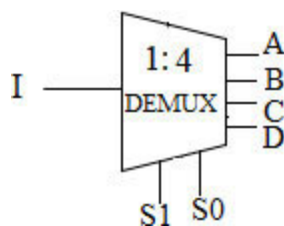


Figure 5.8: 1 to 4 demultiplexer

The truth [table 5.4](#) depicts the functionality of 1 to 4 demultiplexers. The data input I is connected to output A, when the select line S₁ and S₀ are 0 and data input I is connected to output B when the select line S₁ is 0 and S₀ is 1, the data input I is connected to output C when select line S₁ is 1 and S₀ is 0 and data input I is connected to output D when select lines S₁ and S₀ are 1. The demultiplexer will work when the enable signal is at logic 1, i.e., ON mode.

mode.
mode.
mode.
mode.
mode.

Table 5.4: *The truth table of 1 to 4 demultiplexer*

The expression for outputs can be written as:

$$A = I \cdot \overline{S1} \cdot \overline{S0}$$

$$B = I \cdot \overline{S1} \cdot S0$$

$$C = I \cdot S1 \cdot \overline{S0}$$

$$D = I \cdot S1 \cdot S0$$

From these expressions, a 1 to 4 demultiplexer can be implemented using logic gates, as per [Figure](#)

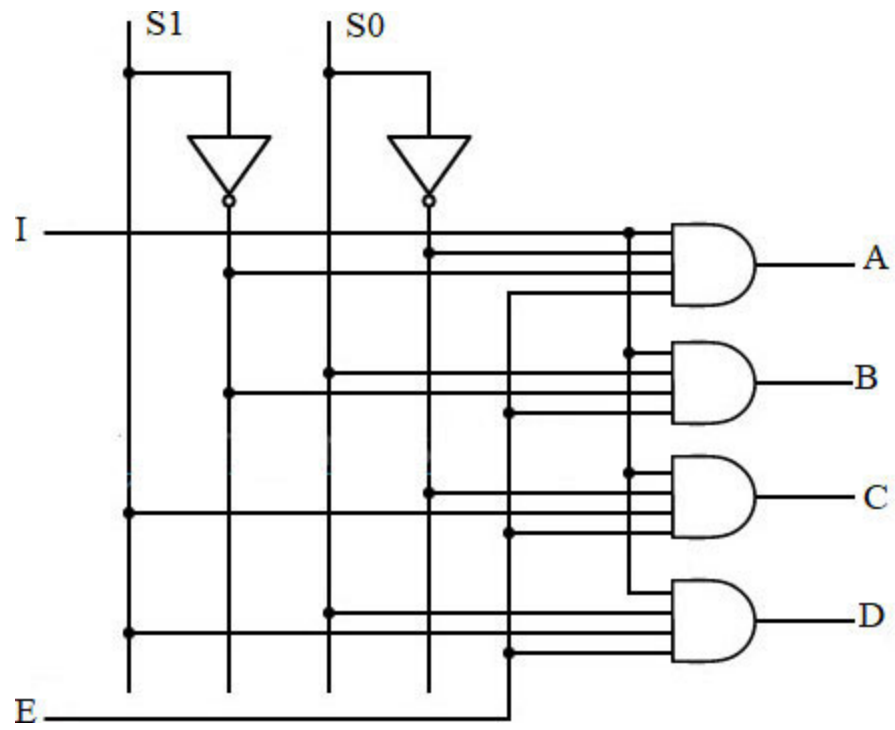


Figure 5.9: Logic diagram of 1 to 4 demultiplexer

5.9. Verilog_program for 1:2 demux design

The demultiplexer is implemented using data flow modelling, gate level modelling and behavioral modelling.

5.9.1. Demultiplexer implementation using dataflow modelling

```
module DeMUX_data(i,s,y0,y1);  
input i,s;  
output y0,y1;  
assign y0=i&(~s);  
assign y1=i&s;  
endmodule
```

5.9.2. Demultiplexer implementation using gate-level modelling

```
module DeMUX_gate(i,s,y0,y1);  
input i,s;  
output y0,y1;  
and a1(y0,i,(~s));  
and a2(y1,i,s);  
endmodule
```

5.9.3. Demultiplexer implementation using behavioral modelling

```
module DeMUX_behav(i,s,y0,y1);  
input i,s;  
output reg y0,y1;  
always@(*)  
begin  
if(!s)  
y0=i;  
else  
y1=i;  
end  
endmodule
```

5.10. Verilog_program for 1:4 demux design

The demultiplexer is implemented using data flow modelling, gate level modelling and behavioral modelling.

5.10.1. Demultiplexer implementation using dataflow modelling

```
module DeMUX14_data(i,s,y);  
input i;  
input [1:0]s;  
output [3:0]y;  
assign y[0]=i&(~s[1])&(~s[0]);  
assign y[1]=i&(~s[1])&s[0];  
assign y[2]=i&s[1]&(~s[0]);  
assign y[3]=i&s[1]&s[0];  
endmodule
```

5.10.2. Demultiplexer implementation using gate-level modelling

```
module DeMUX14_data(i,s,y);
input i;
input [1:0]s;
output reg [3:0]y;
always@(*)
begin
if(s=2'boo)
begin
y[0]=i; y[3:1]=3'b000;
end
else if(s=2'bo1)
begin
y[1]=i; y[3:2]=2'b00; y[0]=1'bo;
end
else if(s=2'b10)
begin
y[2]=i; y[1:0]=2'b00; y[3]=1'bo;
end
else
begin
y[3]=i; y[2:0]=3'b000;
end
end
```



```
end  
endmodule
```

5.11. Verilog_program for 1:8 demux design

The demultiplexer is implemented using behavioral modelling.

5.11.1. Demultiplexer implementation using behavioral modeling

```
module DeMUX18_data(i,s,y);
input i;
input [2:0]s;
output reg [7:0]y;
always@(*)
begin
case(s)
3'b000: begin y[0]=i; y[7:1]=7'b00000000; end
3'b001: begin y[1]=i; y[7:2]=6'b0000000; y[0]=1'bo; end //vector
part select
3'b010: begin y[2]=i; y[7:3]=5'b000000; y[1:0]=2'boo; end
3'b011: begin y[3]=i; y[7:4]=4'b00000; y[2:0]=3'b000; end
3'b100: begin y[4]=i; y[7:5]=3'b000; y[3:0]=4'b0000; end
3'b101: begin y[5]=i; y[7:6]=2'boo; y[4:0]=5'b00000; end
3'b110: begin y[6]=i; y[7]=1'bo; y[5:0]=6'b000000; end
3'b111: begin y[7]=i; y[6:0]=7'b0000000; end
Endmodule
```

Conclusion

In this chapter, we studied the significance and design of combinational circuits using multiplexer and de-multiplexer. The implementation of functions is discussed using Verilog HDL.

In the next chapter, we will study the design and implementation of combinational logic circuits using encoder and decoder.

Questions

Difference between multiplexer and de-multiplexer.

Design using k-map.

Implement 4:1 multiplexer using conditional operators.

Implement 1:8 de-multiplexer using structural modelling.

Design and implement the following equation using Verilog HDL

CHAPTER 6

Encoder/Decoder Implementation Using Verilog HDL

6.1. Introduction to encoder

The encoder is a combinational circuit that converts data information from 2^N input lines to N output lines. An encoder is used to convert an analog signal into a digital signal, such as a binary coded decimal form. The generalized block diagram of the encoder is, as shown in [Figure](#)

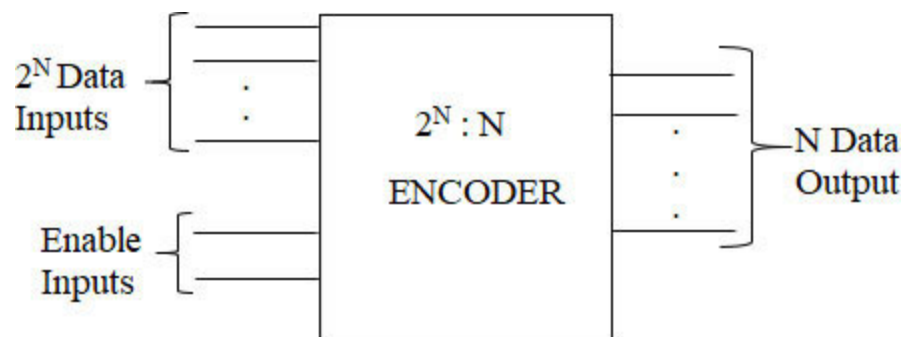


Figure 6.1: Block diagram of an encoder

Structure

In this chapter, we will discuss the following topics:

Significance of encoder and decoder

Implementation of encoder and decoder using Verilog HDL

Objectives

After studying this unit, you should be able to:

Understand the design of combinational logic circuits using encoder and decoder.

Implementation of various encoders and decoders using Verilog HDL.

6.2. Octal to binary encoder

In octal to the binary encoder, also known as 8 to 3 encoder, there will be eight inputs and three outputs.

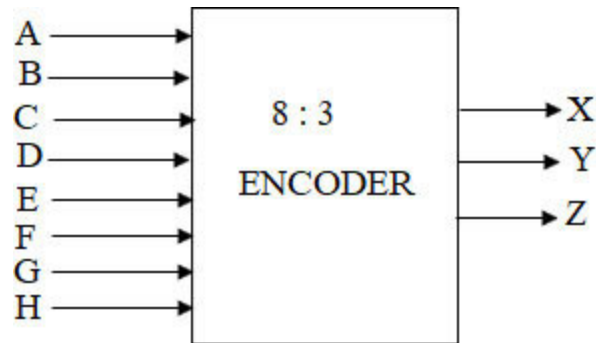


Figure 6.2: 8 to 3 encoder

The truth [table](#) depicts the basic understanding of 8 to 3 encoder concept and output equations are framed.

INPUTS								OUTPUTS		
A	B	C	D	E	F	G	H	X	Y	Z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

$X=E+F+G+H$

$Z=B+D+F+H$

$Y=C+D+G+H$

Table 6.1: The truth table of octal to binary encoder

From the truth table, the output equations can be framed by looking into the input table. The placement of '1' in the input table corresponding to '1' value of output gives the expression for output equation.

$$X=E+F+G+H$$

$$Y=C+D+G+H$$

$$Z=B+D+F+H$$

The logical diagram of octal to the binary encoder is represented as in [Figure](#)

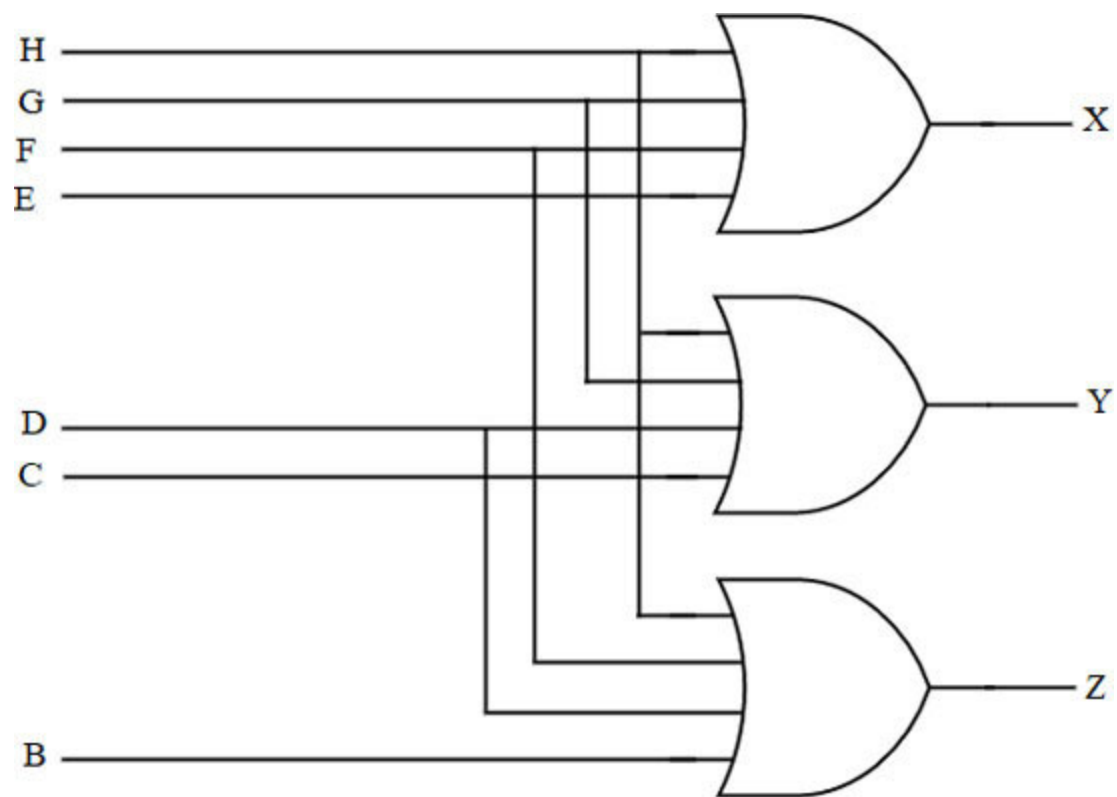


Figure 6.3: Logic diagram of 8 to 3 encoder

6.3. Decimal to BCD encoder

In decimal to BCD encoder, there will be ten input lines, and four will be selected as output lines.

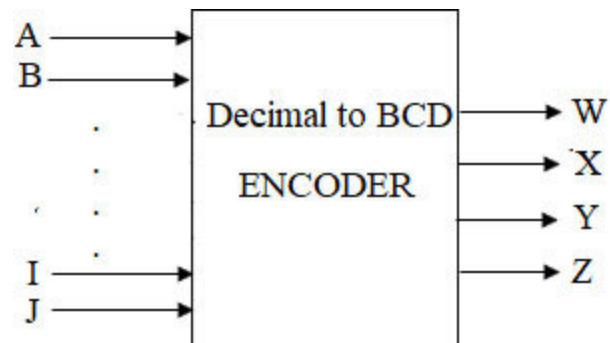


Figure 6.4: *Decimal to BCD encoder*

The truth [table](#) depicts the basic understanding of decimal to BCD encoder concept, and output equations are framed.

framed. framed.
framed.
framed.
framed.

framed.
framed.
framed.
framed.
framed.
framed.
framed.
framed.
framed.

Table 6.2: *The truth table of decimal to BCD encoder*

From the truth table, the output equations can be framed by looking into the input table. The corresponding value of the input for the placement of ‘1’ in output, gives the expression for output equation.

$$W=8+9$$

$$X=4+5+6+7$$

$$Y=2+3+6+7$$

$$Z=1+3+5+7+9$$

The logical diagram of decimal to BCD encoder is represented as in [Figure](#)

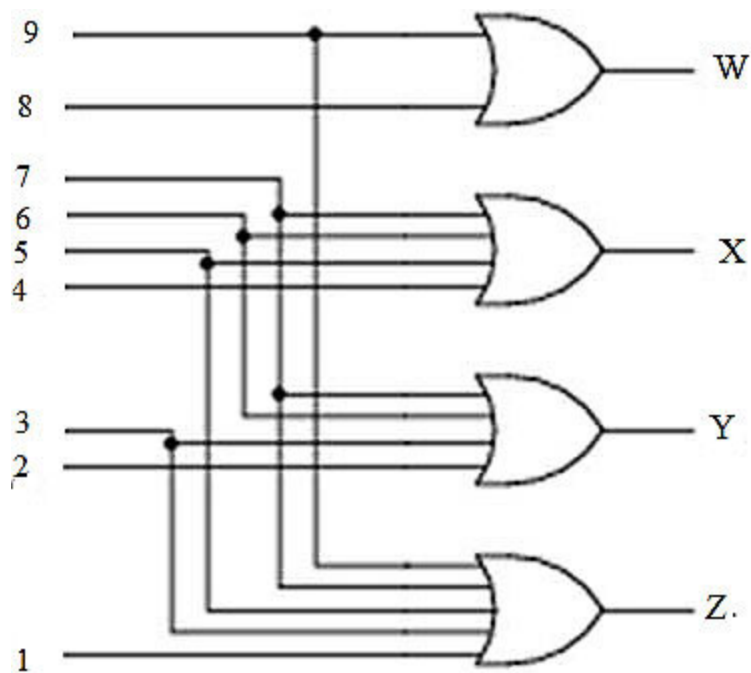


Figure 6.5: Logic diagram of decimal to BCD encoder

6.4. Verilog_program for 4:2 encoder

The encoder is implemented using data flow modeling and behavioral modeling.

6.4.1. The 4:2 encoder design using dataflow modeling

The data flow model is based on logic equations and keyword assign. It is used in writing the code for Boolean expressions. At this level, the module is designed by specifying the data flow.

```
module encoder42_data(a,y);  
input [3:0]a;  
output [1:0]y;  
assign y[0]=a[3]|a[1];  
assign y[1]=a[3]|a[2];  
endmodule
```

6.4.2. The 4:2 encoder design using behavioral modeling

Behavioral modeling is related to the algorithm or behavior of the circuit. So, it is also known as algorithmic modeling. The behavioral modelling is said to be the highest level of abstraction provided by Verilog HDL as it reduces the design steps & minimize the efforts of the designer.

```
module encoder42_behav(a,y);  
input [3:0]a;  
output reg[1:0]y;  
always@(*)  
begin  
case(a)  
4'b1000:y=2'b11;  
4'b0100:y=2'b10;  
4'b0010:y=2'b01;  
4'b0001:y=2'b00;  
endcase  
end  
endmodule
```

6.5. Priority encoder

The priority encoder is an encoder circuit which generates wrong code, in case more than one input is active. It works along with priority function. A level is assigned to each input to resolve the problem of priority encoder; if more than one input is active at the same time, the output will explore the levels of inputs and selects the input, whose subscript is having the largest numerical value. It is used to compress multiple inputs into the lesser number of outputs. The truth table for priority encoder is illustrated in [table](#)

Table 6.3: *The truth table of 4:2 priority encoder*

The 'X' designates an unknown number; it means the number may be '0' or '1'. The Input I₃ has the highest priority, so whenever I₃ is high, the output will be '1' irrespective of the values of other inputs. An indicator parameter 'P' is there, which accesses the priority level. If all inputs are '0', then the

indicator 'P' is 'o', and when one or more inputs are equal to '1', the indicator 'P' will be '1'. The expression of outputs can be explored using a Karnaugh map.

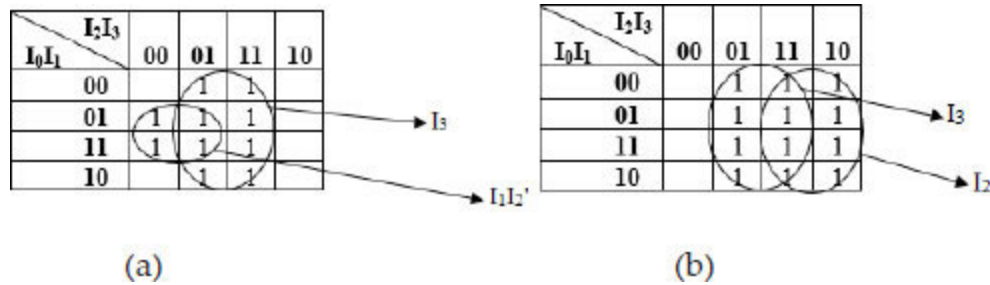


Figure 6.6: K-map for output E (a) and F(b)

The expression for E, F, and P are:

$$E = I_3 + I_1 I_2', \quad F = I_2 + I_3 \quad \text{and} \quad P = I_0 + I_1 + I_2 + I_3$$

The logic diagram for priority encoder is:

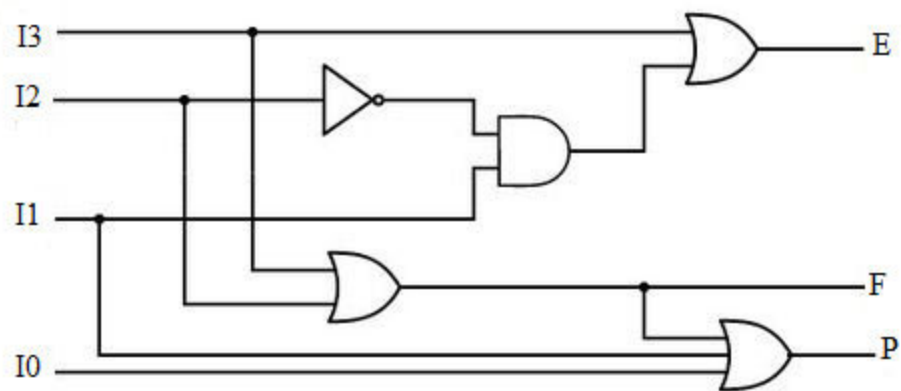


Figure 6.7: *Logic diagram for 4:2 priority encoder*

6.6. Verilog code for priority encoder design

```
module encoder42_priority(a,y,val);
input [3:0]a;
output reg[1:0]y;
output reg val;
always@(*)
begin
case(a)
4'b1000: begin y=2'b11; val=1'b1; end
4'bx100: begin y=2'b10; val=1'b1; end
4'bxx10: begin y=2'b01; val=1'b1; end
4'bxxx1: begin y=2'b00; val=1'b1; end
default: begin y=2'bxx; val=1'bo; end
endcase
end
endmodule
```

6.7. Introduction to decoder

An n -bit binary code has distinct 2^n units. The decoder is a combinational circuit that detects a particular digital state. It is used to convert n input binary numbers to 2^n output binary numbers; here, n is a number of bits. The block diagram of the decoder is, as shown in [figure](#)

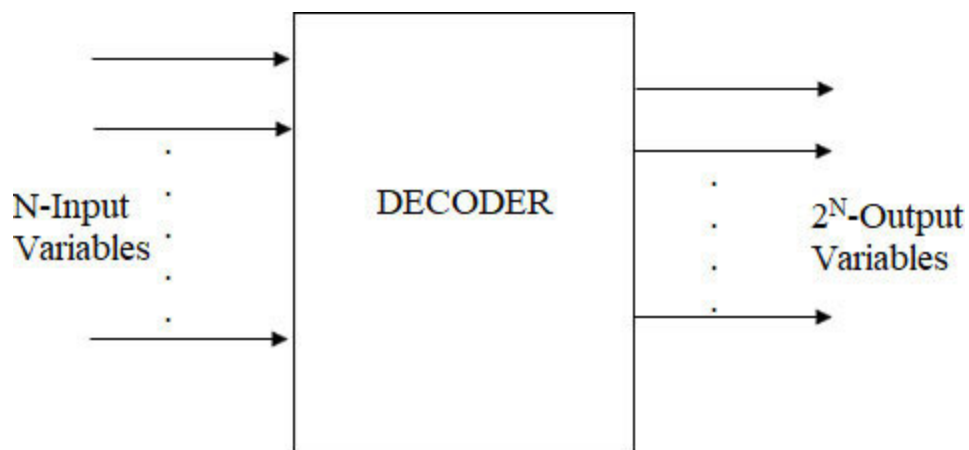


Figure 6.8: Block diagram of a decoder

6.8. The 2 line to 4 line decoder

In 2 to 4 decoder, the number of input variables is two, and the output variables are The block diagram of two to four decoder is shown in [figure](#)

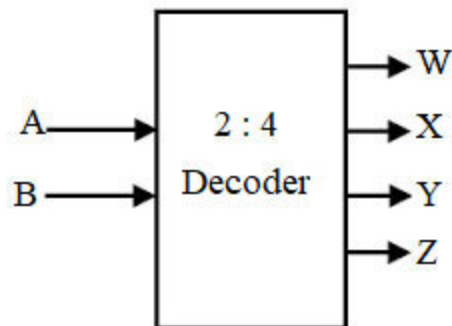


Figure 6.9: Block diagram of 2 to 4 decoder

The truth table of 2 to 4 decoder is depicted in [table](#) where inputs and outputs are displayed. The decoder will work when the enable line is active (logic 1).

1).
1).
1).

1).
1).
1).
1).

Table 6.4: The truth table of 2 to 4 decoder

When the enable is '0', irrespective of the value of input A and B, the output is always '0'. The decoder will work when the enable is '1'. The output 'Z' will active when the input is $AB=00$, the output 'Y' will active, when the input is $AB=01$, similarly when the input AB is 10 or 11, the output 'X' or 'W' will be activated respectively. A decoder will enable input can function like a demultiplexer, with the difference that a number of inputs are 1 in the case of the demultiplexer. The logic diagram for 2 to 4 decoder is shown as in [figure](#)

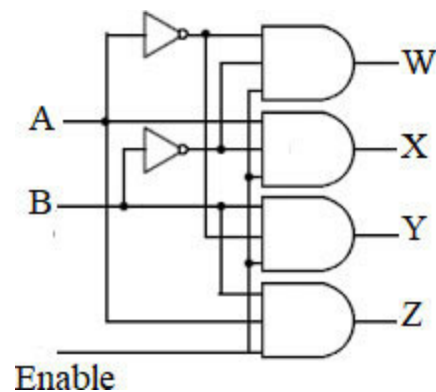


Figure 6.10: *Logic diagram for 2 to 4 decoder*

6.9. The 3 line to 8 line decoder

In 3 to 8 decoder, the number of input variables is three, and the output variables are The block diagram of three to eight decoder is shown as in [figure](#)

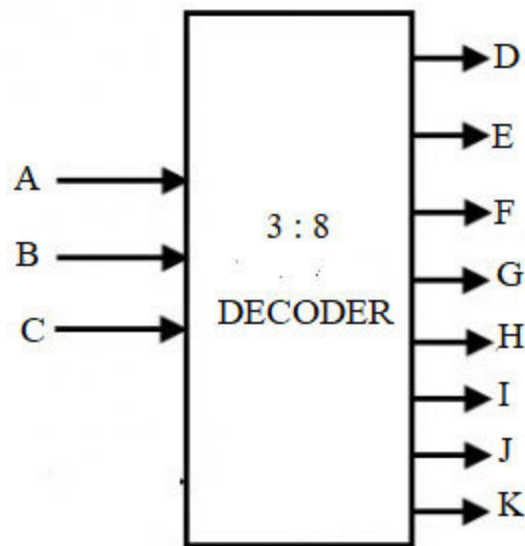


Figure 6.11: Block diagram of 3 to 8 decoder

The truth table of 3 to 8 decoder is depicted in [table](#) where inputs and outputs are displayed. The decoder will work when the enable line is active (logic 1).

1).
1).
1).
1).
1).

1).
1).
1).
1).
1).

Table 6.5: Truth table of 3 to 8 decoder

When the enable is '0', irrespective of the value of input A, B, and C, the output is always '0'. The decoder will work when the enable is '1'. The output 'D' will active, when the input is ABC=000, the output 'E' will active, when the input is ABC=001, similarly when the input ABC is 010, output 'F' will active, when ABC=011, the output 'G', at ABC=100, the output 'H', at ABC=101, the output 'I' and at input ABC=110, the output 'J' will be activated respectively. At input ABC=111, the output 'K' will be high. A decoder will enable input can

function like a demultiplexer, with the difference that a number of inputs are 1 in case of the demultiplexer. The logic diagram for 3 to 8 decoder is shown in [figure](#)

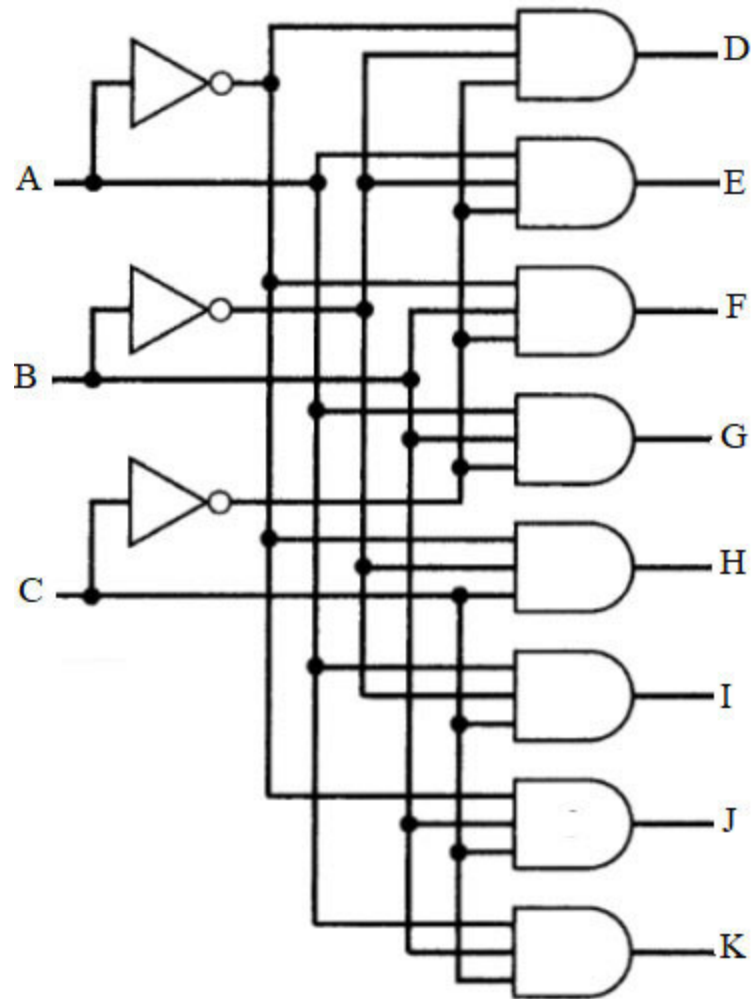


Figure 6.12: Logic diagram for 3 to 8 decoder

6.10. Verilog_program for 2:4 decoder design

The decoder is implemented using data flow modelling, gate level modelling and behavioral modelling.

6.10.1. Decoder implementation using dataflow modelling

The dataflow model is based on logic equations and keyword assign. It is used in writing the code for Boolean expressions.

```
module Decoder_data(a,y);  
input [1:0]a;  
output [3:0]y;  
assign y[0]=(~a[1])&(~a[0]);  
assign y[1]=(~a[1])&a[0];  
assign y[2]=a[1]&(~a[0]);  
assign y[3]=a[1]&a[0];  
endmodule
```

6.10.2. Decoder implementation using gate-level modelling

In gate level modeling, the design can be implemented using logic gates and Verilog HDL has a predefined set of logic gates. As this style is closer to physical implementation, so, gate level modeling is also known as structural modeling.

```
module Decoder_gate(a,y);  
input [1:0]a;  
output [3:0]y;  
and a1(y[0],(~a[1]),(~a[0]));  
and a2(y[1],(~a[1]),a[0]);  
and a3(y[2],a[1],(~a[0]));  
and a4(y[3],a[1],a[0]);  
endmodule
```


6.10.3. Decoder implementation using behavioral modelling

Behavioral modeling is related to the algorithm or behavior of the circuit. So, it is also known as algorithmic modeling. This is the highest level of abstraction provided by Verilog HDL.

```
module Decoder_behav(a,y);
input [1:0]a;
output reg [3:0]y;
always@(*)
begin
case(a)
2'b00: y=4'b0001;
2'b01: y=4'b0010;
2'b10: y=4'b0100;
2'b11: y=4'b1000;
endcase
end
endmodule
```

6.11. Verilog_program for 3:8 decoder design

The decoder is implemented using data flow modelling, gate level modelling and behavioral modelling.

6.11.1. Decoder implementation using behavioral modelling

```
module Decoder38_behav(a,b,c,y);
input a,b,c;
output reg [7:0]y;
always@(*)
begin
case({a,b,c})
3'b000: y=8'b00000001;
3'b001: y=8'b00000010;
3'b010: y=8'b00000100;
3'b011: y=8'b00001000;
3'b100: y=8'b00010000;
3'b101: y=8'b00100000;
3'b110: y=8'b01000000;
3'b111: y=8'b10000000;
endcase
end
endmodule
```

6.11.2. The 3:8 decoder implementation using 2:4 decoder

```
module DecoderEN_behav(en,a,y);
input [1:0]a;
input en;
output reg [3:0]y;
always@(*)
begin
if(en)
begin
case(a)
2'b00: y=4'b0001;
2'b01: y=4'b0010;
2'b10: y=4'b0100;
2'b11: y=4'b1000;
endcase
end
else
y=4'b0000;
end
endmodule
module Decoder38_behav(a,y);
input [2:0]a;
output [7:0]y;
```

```
DecoderEN_behav DN1((~a[2]),a[1:0],y[3:0]); //3:8 decoder using  
2:4 decoder  
DecoderEN_behav DN2(a[2],a[1:0],y[7:4]);  
endmodule
```

Conclusion

In this chapter, we studied the design and implementation of various types of encoders and decoders using Verilog HDL.

In the next chapter, we will study the magnitude comparator and its implementation using Verilog HDL.

Questions

Differentiate between encoder and decoder.

How encoder is different than multiplexer.

How the decoder is different than de-multiplexer.

Design a 1-bit full adder using the decoder.

Design 1-bit full subtractor using the decoder.

CHAPTER 7.

Magnitude Comparator Implementation Using Verilog HDL

7.1. Introduction

A digital comparator is used to compare the digital signals at the input terminal and produces an output, as per the outcome of the comparison. Mainly, two types of digital comparators are there to compare binary bits.

Identity comparator: This is one type of digital comparator, where the output terminal is single. For example, A and B are two binary numbers, whose identity needs to compare. It will compare A with B, for condition $A=B$ and produces output 1 (true) ($A=B=0$ or $A=B=1$) or 0 (false) ($A \neq B$), as per the comparison output. Only one type of comparison ($A=B$) is done with this comparator.

Magnitude In this type of digital comparator, there are three output terminals. It will compare binary bits A and B for conditions equal to, greater than and less than, i.e., $A=B$, $A > B$, and $A < B$. Three types of comparisons are performed using a magnitude comparator. The general specification of the n-bit comparator is shown in [figure](#)

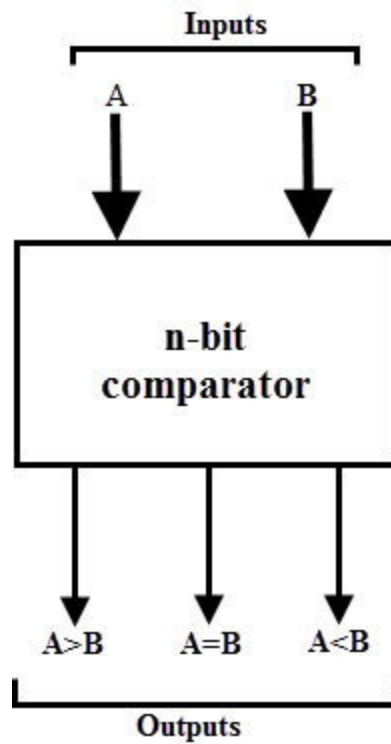


Figure 7.1: n-bit comparator

Digital comparators use the exclusive-NOR gate in their design and operation because the comparator will generate output 1 when two inputs are equal.

Structure

In this chapter, we will discuss the following topics:

Need and significance of the comparison

Design and implementation of magnitude comparator using Verilog HDL.

Objectives

After studying this unit, you should be able to:

Design the one-bit and two-bit magnitude comparator using K-map

Implementation of magnitude comparator using Verilog HDL

7.2. One-bit magnitude comparator

The one-bit magnitude comparator performs its operation when the input consists of one bit. For example, A and B are two one-bit binary numbers, and the comparator will check all the three conditions: equal to, greater than and less than.

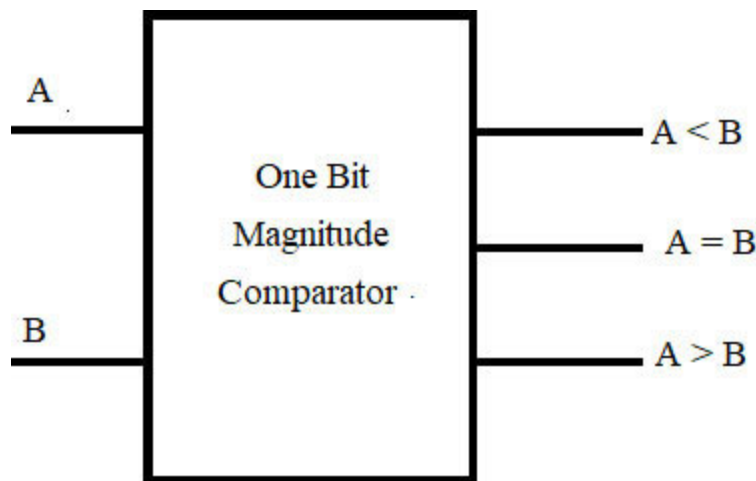


Figure 7.2: Block diagram of the one-bit magnitude comparator

[Figure 7.2](#) represents the block diagram of a one-bit magnitude comparator. As the number of inputs is two, the possible combinations will be four. The truth [table](#) depicts the comparison of A and B.

B.
B.
B.
B.
B.
B.

Table 7.1: One-bit magnitude comparator

Verilog program:

```

module Mag_Comp2bit(aGRTb, aEQUb, aLESb,a,b);
input a,b;
output aGRTb, aEQUb, aLESb;
assign aGRTb=A&(~B);
assign aLESb=A~^B;
assign aLESb= (~A)&B;
endmodule

```

7.2.1. One bit magnitude comparator using logic gates

The Boolean expression can be explored by using a Karnaugh map. There will be three k-maps for output, as shown in [figure](#)

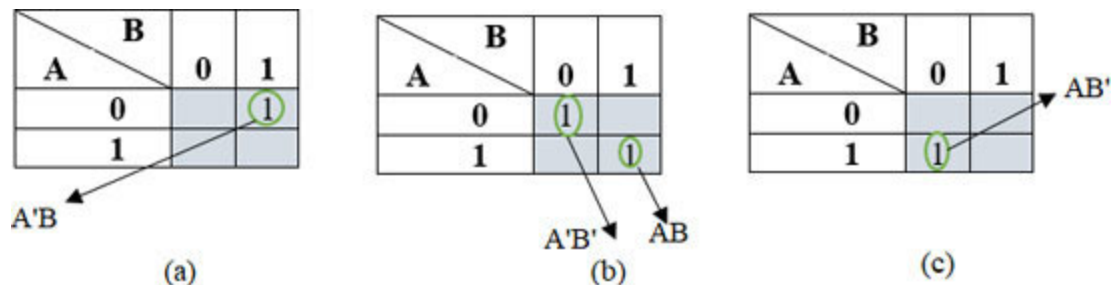


Figure 7.3: K-map for (a) $A=B$ (b) $A>B$ (c) $A<B$

So, the Boolean expression for $A=B$ is and $A>B$ is

The logic diagram for a one-bit magnitude comparator is shown in [figure](#)

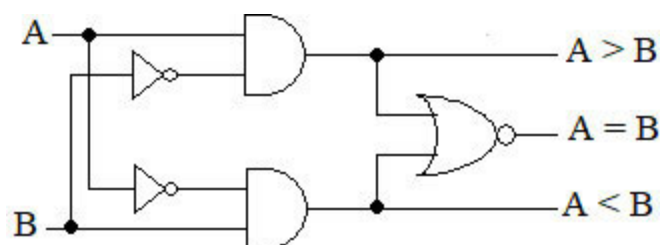


Figure 7.4: Logic diagram of the one-bit magnitude comparator

7.3. Two-bit magnitude comparator

The two-bit magnitude comparator performs its operation when the input consists of two bits. For example, A and B are two-bit binary numbers, and the comparator will check all the three conditions: equal to, greater than and less than.

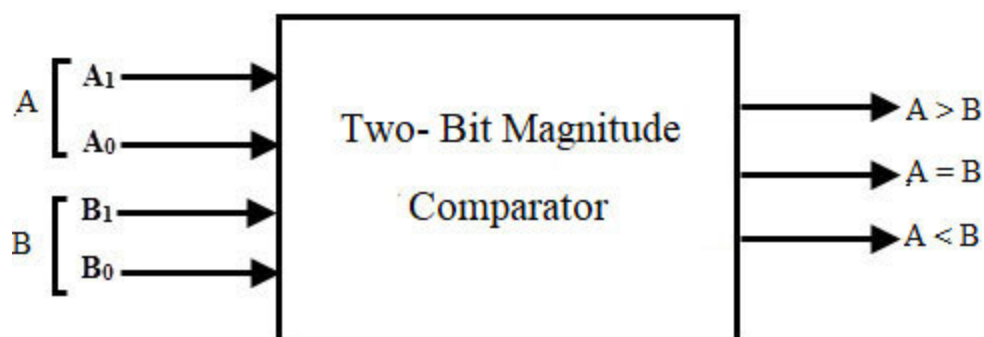


Figure 7.5: Block diagram of the one-bit magnitude comparator

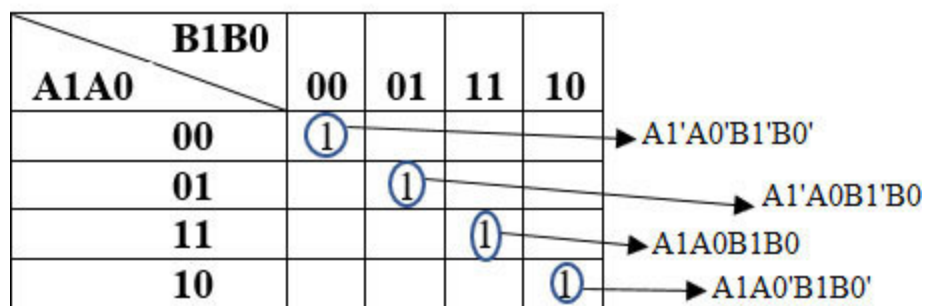
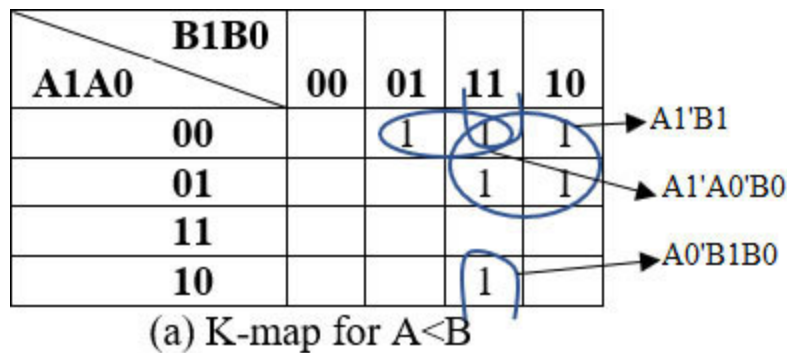
Figure 7.5 represents the block diagram of a one-bit magnitude comparator. As each binary number consists of two bits, all the possible combinations are shown in [table](#)

[illegible]

Table 7.2: Two-bit magnitude comparator

7.3.1. Two-bit magnitude comparator using logic gates

The Boolean expression can be explored by using a Karnaugh map. There will be three k-maps for output, as shown in [figure](#)



(b) K-map for $A=B$

$\begin{matrix} \text{B1B0} \\ \text{A1A0} \end{matrix}$		00	01	11	10
		00	01	11	10
	00				
	01	1			
	11	1	1		1
	10	1	1		

$\rightarrow A0B1'B0'$
 $\rightarrow A1A0B0'$
 $\rightarrow A1B1'$

(c) K-map for $A>B$

Figure 7.6: K-map for (a) A (b) $A=B$ (c) $A>B$

The logic diagram for the two-bit magnitude comparator is shown as in [figure](#)

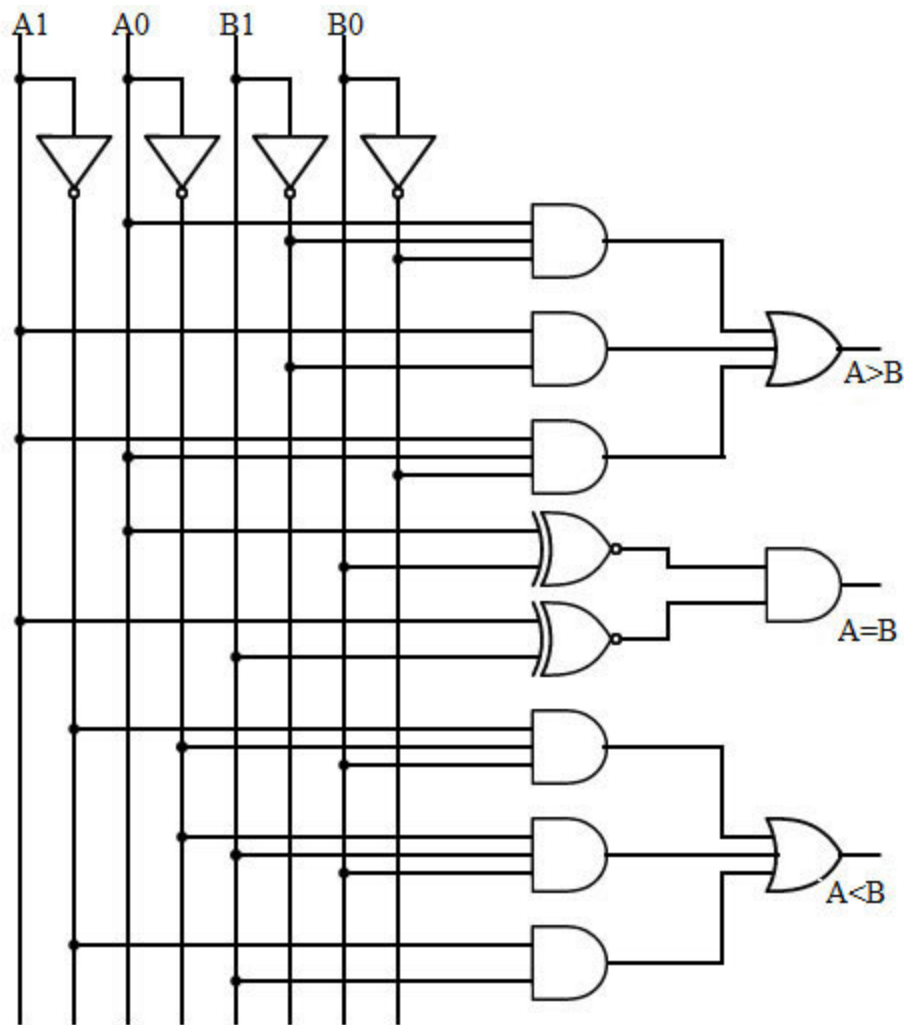


Figure 7.7: Logic diagram of the two-bit magnitude comparator

7.3.2. N-bit magnitude comparator using Verilog HDL

```
module Mag_CompNbit(aGRTb, aEQUb, aLESb,a,b);
input [7:0]a,b;
output reg aGRTb, aEQUb, aLESb;
always @(*)
begin
if(a>b)
begin
aGRTb=1'b1;
aEQUb=1'b0;
aLESb=1'b0;
end
else if(a
begin
aGRTb=1'b0;
aEQUb=1'b0;
aLESb=1'b1;
end
else if(a==b)
begin
aGRTb=1'b0;
aEQUb=1'b1;
aLESb=1'b0;
```

```
end
else
begin
aGRTb=1'bx;
aEQUb=1'bx;

aLESb=1'bx;
end
end
endmodule
```

Conclusion

In this chapter, we studied the design and implementation of the magnitude comparator, where the decision is in terms of binary output '0' or '1'.

In the next chapter, we will study the various types of flip-flops and its implementation using Verilog HDL.

Questions

What is the significance of the term 'magnitude' in magnitude comparator?

How many inputs are required for a digital comparator?

Differentiate between one-bit and two-bit magnitude comparator?

Design a two-magnitude comparator using a conditional operator?

CHAPTER 8

Flip-Flop Implementation Using Verilog HDL

8.1. Introduction to flip-flop

A latch is an asynchronous sequential circuit, where any change in input can be seen on the output immediately. Further, to modify the operation of latch, an extra control signal is associated with a latch that controls the transition of the input signal. This extra control signal is called a clock or clock pulse. A latch with a clock is known as a flip-flop.

Structure

In this chapter, we will discuss the following topics:

Significance and types of flip-flop

Characteristics and excitation table of all the flip-flops

Design and implementation of flip-flop using Verilog HDL

Objectives

After studying this unit, you should be able to:

Understand the working of all the flip-flops.

Convert the one type of flip-flop into another.

Design and implementation of sequential circuits using flip-flop and Verilog HDL.

8.1.1. Types of flip-flop

Flip-flop is also known as a bi-stable latch because when the circuit is triggered by an input pulse, it will remain in that state unless further input pulse is not applied. So, the circuit remembers that state, so it is also known as a bi-stable latch. Based on the operation and process, there are four different types of flip-flops:

SR flip-flop

JK flip-flop

D flip-flop

T flip-flop

8.2. SR flip-flop

The SR flip-flops are also known as 1-bit memory because even after the input pulse, the value of the input pulse is stored in SR flip-flop. The circuit will work only when the clock is activated. Further, its working is based on definite edge triggering or negative edge triggering. The symbol of SR flip-flop is shown in [Figure](#)

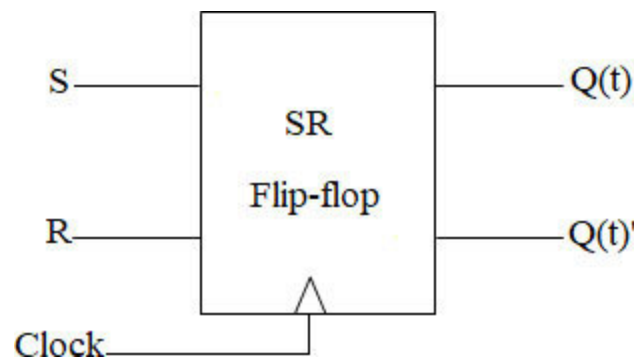


Figure 8.1: Symbol of SR flip-flop

The SR flip-flop can be constructed using NAND or NOR gate because both gates are having the capability of a universal gate. [Figure 8.2](#) shows SR flip-flop using NAND gates.

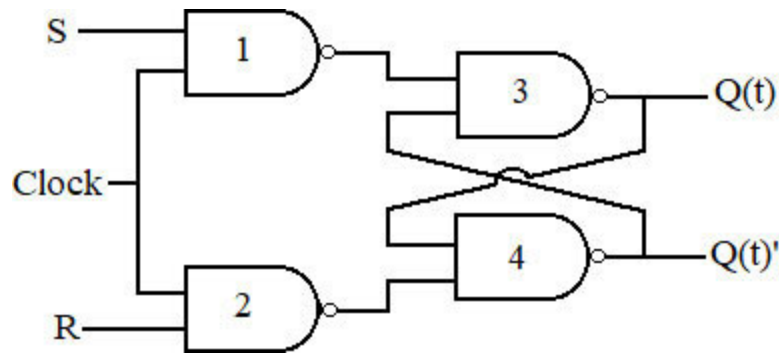


Figure 8.2: SR flip-flop using NAND gates

The clock pulse plays a vital role in the operation of flip-flops. When the clock is absent, the circuit will retain the same state.

Case (I): $R=0$ and $S=0$

Let us assume $Q(t)=1$ and $Q(t)'=0$, $\text{Clock}=1$

The output of gate1 and gate2 will be 1. So, the output of gate3 will be 1, and gate4 output will be 0. It means the same previous value is again at the output terminal. This condition is known as 'no change' state.

Case (II): $R=1$ and $S=0$, $\text{Clock}=1$

The output of gate1 will be 1, and the output of gate2 will be 0. In the NAND gate, if any of the inputs are at a low level, i.e., 0, then output is 1. So, the output of gate3, i.e., $Q(t)=0$ and $Q(t)'=1$. This is known as a reset state.

Case (III): $R=0$ and $S=1$, $Clock=1$

The output of gate1 will be 0, and the output of gate2 will be 1. In the NAND gate, if any of the inputs are at a low level, i.e., 0, then output is 1. So, the output of gate3, i.e., $Q(t)=1$ and $Q(t)'=0$. This is known as a set state.

Case (IV): $R=1$ and $S=1$, $Clock=1$

The output of both gate1 and gate2 will be 0. In the NAND gate, if any of the inputs are at a low level, i.e., 0, then output is 1. So, the output of gate3, i.e., $Q(t)=0$ and $Q(t)'=0$. This is known as a prohibited state or indeterminate state because both the outputs should be complementary. [Table 8.1](#) shows the input and output relationship for SR flip-flop using a NAND gate.

gate.
gate. gate.

gate. gate.
gate.
gate.
gate.

Table 8.1: The truth table of SR flip-flop using NAND gate

8.2.1. Characteristics table of SR flip-flop

A characteristics function or equation of flip-flop is used to explore the value of the next state in terms of current state and output. It defines the logical property of flip-flops. [Table 8.2](#) shows the characteristics properties of SR flip-flop.

flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.

Table 8.2: Characteristics table of SR flip-flop

The characteristics table is filled based on the truth table. For example, if $S=0$, $R=0$, there is no change. The same is applicable in [table](#) where the value of the previous state will be transferred to the next state, without any change. Similarly, all other values are executed. The compiled table for SR flip-flop is shown as in [table](#)

--

Table 8.3: Summarized characteristics table of SR flip-flop

The characteristics equation can be explored using k-map, as in the [Figure](#)

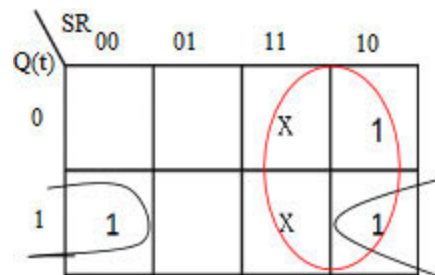


Figure 8.3: K-map of SR flip-flop characteristics table

8.2.2. Excitation table of SR flip-flop

An excitation table of flip-flop shows the necessary input conditions for every possible transition of output. Based on the present and next state values, the input conditions are framed. The symbol 'X' signifies don't care condition, which means that it does not matter whether the input is 0 or 1. [Table 8.4](#) shows the excitation table of SR flip-flop.

flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.

Table 8.4: Excitation table of SR flip-flop

The excitation table is deduced from the characteristics table. The characteristics table is looked for the values when $Q(t)=0$ and $Q(t+1)=0$, the corresponding values for this condition are $S=0$, $R=0$, and $S=0$, $R=1$, which can be written as summarised

from as: $S=0$, $R=X$. Similarly, the other values of the excitation table of SR flip-flops are written.

Verilog program

```
module srff(q,clk,rst,s,r);
input clk,rst,s,r;
output reg q=0;
always@(posedge clk or posedge rst)
begin
if(rst)
q=0;

else if(!s&&!r)
q=q;
else if (!s&&r)
q=1'b0;
else if (s&&!r)
q=1'b1;
else
q=1'bx;
end
endmodule
```

8.3. JK flip-flop

The problem with SR flip-flop is its undetermined state at values $S=1$ and $R=1$. JK flip-flop is refined flipflop, in which an undetermined state of SR flip-flop is defined. The symbol of the JK flip-flop is shown in the [Figure](#) where inputs are J and K with corresponding outputs $Q(t)$ and $Q(t)'$. The JK flip-flop is a combination of SR flip-flop having feedback. The letters J and K are chosen by inventor Jack Kirby.

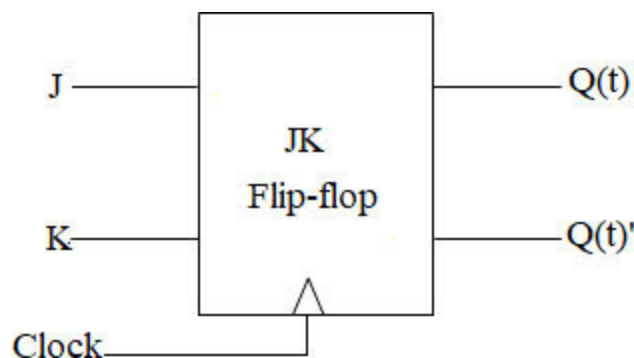


Figure 8.4: Symbol of JK flip-flop

The JK flip-flop can be constructed using NAND or NOR gate because both gates are having the capability of the universal gate. The [Figure](#) shows the JK flip-flop using NAND gates.

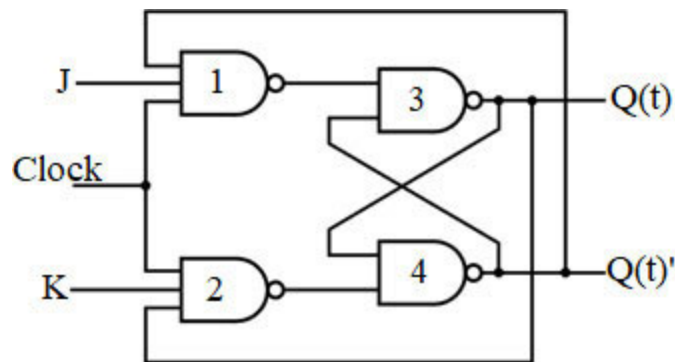


Figure 8.5: JK flip-flop using NAND gates

The clock pulse plays an important role in the operation of flipflops. When the clock is absent, the circuit will retain the same state.

Case (I): $J=0$ and $K=0$

Let us assume $Q(t)=1$ and $Q(t)'=0$, $\text{Clock}=1$

The output of gate1 and gate2 will be 1. So, the output of gate3 will be 1, and gate4 output will be 0. It means the same previous value is again at the output terminal. This condition is known as 'no change' state.

Case (II): $J=0$ and $K=1$

When $J=0$ and $K=1$, then the output of gate1 and gate2 will high. The value of $Q(t)'$ will be fed to gate3, and the calculated value will be fed to gate4; this implies $Q(t)=0$, i.e., reset condition.

Case (III): $J=1$ and $K=0$

When $J=1$ and $K=0$, then the output of gate1 will be 0 (assume $Q(t)'=1$), and gate2 will be 1. The value of $Q(t)'$ will be fed to gate3, and the calculated value will be fed to gate4; this implies $Q(t)=1$, i.e., set condition.

Case (IV): $J=1$ and $K=1$

When $J=1$ and $K=0$, then the output of gate1 will be 0 (assume $Q(t)'=1$), and gate2 will be 1. The value of $Q(t)'$ will be fed to gate3, and the calculated value will be fed to gate4. When the assumed values of $Q(t)=0$ and $Q(t)'=1$, then after execution of feedback circuits, the output value of $Q(t) = 1$ and $Q(t)'=0$. Similarly, When the assumed values of $Q(t)=1$ and $Q(t)'=0$, then after execution of feedback circuits, the output value of $Q(t) = 0$ and $Q(t)'=1$. Hence, the toggle condition will exist.

Toggle is an alternate between opposite binary states with the application of clock pulse. The input-output relationship of JK flip-flop is shown as in truth [table](#)

Table 8.5: The truth table of JK flip-flop using NAND gate

8.3.1. Characteristics table of JK flip-flop

A characteristics function or equation of flip-flop is used to explore the value of the next state in terms of current state and output. It defines the logical property of flip-flops. [Table 8.6](#) shows the characteristics properties of JK flip-flop.

flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.

Table 8.6: Characteristics table of JK flip-flop

The characteristics table is filled based on the truth table. For example, if $J=0$, $K=0$, there is no change. The same is applicable in [table](#) where the value of the previous state will be transferred to the next state, without any change. Similarly, all other values are executed.

executed.
executed.
executed.
executed.
executed.

Table 8.7: Summarized characteristics table of JK flip-flop

Using K-map, the characteristics equation can be explored, as in the [Figure](#)

	JK	00	01	11	10
Q(t)					
0			1	1	
1		1			1

Figure 8.6: K-map of JK flip-flop characteristics table

8.3.2. Excitation table of JK flip-flop

An excitation table of flip-flop shows the necessary input conditions for every possible transition of output. Based on the present and next state values, the input conditions are framed. The symbol 'X' signifies don't care condition, which means that it does not matter whether the input is 0 or 1. [Table 8.8](#) shows the excitation properties of JK flip-flop.

flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.

Table 8.8: Excitation table of JK flip-flop

The excitation table is deduced from the characteristics table. The characteristics table is looked for the values when $Q(t)=0$ and $Q(t+1)=0$, the corresponding values for this condition are $J=0$, $K=0$, and $J=0$, $K=1$, which can be written as summarized

from as: $J=0$, $K=X$. Similarly, the other values of the excitation table of JK flip-flops are written.

Verilog program

```
module jkff(q,clk,rst,j,k);
input clk,rst,j,k;
output reg q=0;
always@(posedge clk or posedge rst)
begin
if(rst)
q=0;

else if(!j&&!k)
q=q;
else if (!j&& k)
q=1'b0;
else if (j&&!k)
q=1'b1;
else
q=~q;
end
endmodule
```

8.4. Master-slave JK flip-flop

When $J=1$ and $K=1$ in JK flip-flop for a long duration, the output $Q(t)$ will toggle as long as the clock pulse is high. This condition makes the output of flip-flop unstable. This problem is known as a race-around condition in JK flip-flop. This problem can be avoided by ensuring the factor that the input clock pulse is high for a short duration of time.

The concept of master-slave JK flip-flop is introduced to solve the problem of a race around. Here, two JK flip-flops are cascaded with the help of an inverter, and the feedback channel is also there. One flip-flop act as a master flip-flop and another flip-flop act as a slave flip-flop. The clock pulse will determine which one will be active master flip-flop or slave flip-flop, i.e., if a clock pulse is high, then Master JK flip-flop will work and slave flip-flop will disable because its clock pulse will be low.

8.4.1. Working of a master-slave flip-flop

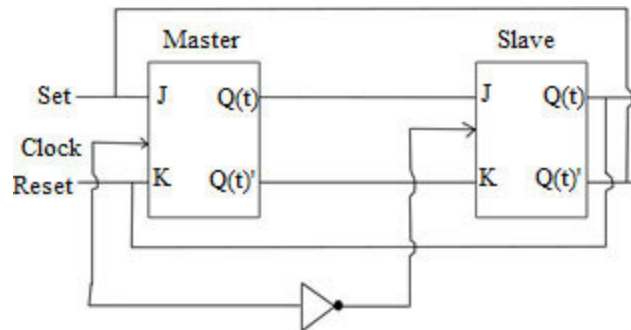


Figure 8.7: Master-slave JK flip-flop

The [figure 8.7](#) shows the master-slave flip-flop. When the clock=1, a slave is disabled until clock=1. When clock=0, the information from master flip-flop is transferred to slave flip-flop, and output is received. The master flip-flop responds well before the slave flip-flop because it is a definite level triggered in comparison with slave flip-flop, which is a negative level triggered.

Case (I): $J=0$ and $K=0$

The flip-flop remains disable and no change in the $Q(t)$.

Case (II): $J=0$ and $K=1$

The high $Q(t)'$ output of the master flip-flop is fed to K input of slave JK flip-flop, and due to clock, the slave is reset. The slave replicates the master flip-flop.

Case (III): $J=1$ and $K=0$

The high $Q(t)$ of a master flip-flop is fed to J input of slave flip-flop, and due to the negative transition of a clock pulse, the slave flip-flop is active, and it copies the master.

Case (IV): $J=1$ and $K=1$

It toggles the transition of the clock pulse. The master flip-flop toggles on positive clock and slave toggles on negative clock transition.

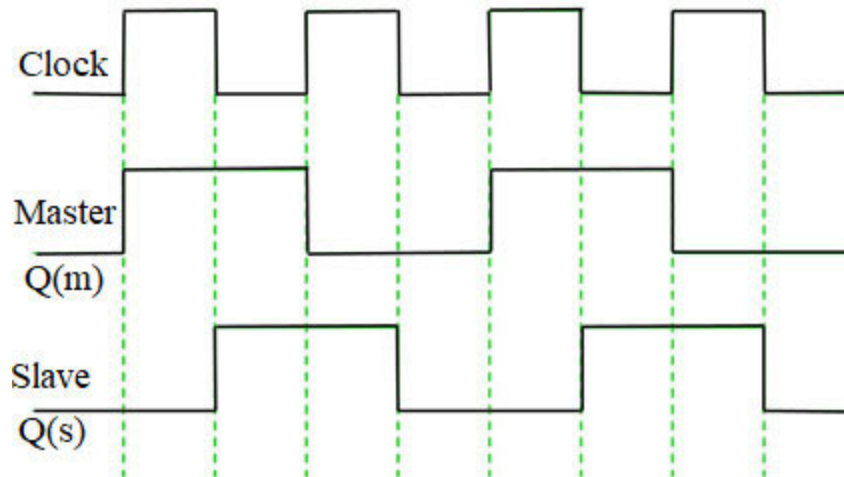


Figure 8.8: The timing diagram of master-slave JK flip-flop

The timing diagram of a master-slave flip-flop is shown in the [figure](#). It is an asynchronous device; it passes data with the transition of a clock pulse.

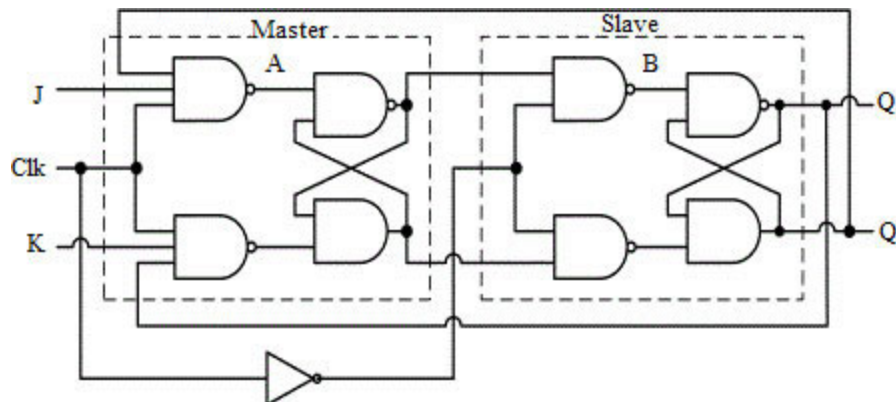


Figure 8.9: Master-slave JK flip-flop using NAND gates

[Figure 8.9](#) shows the circuit diagram of master-slave JK flip-flop using NAND gates.

gates.

gates. gates.
gates.
gates. gates.
gates.
gates. gates.
gates.
gates. gates.
gates. gates.

Table 8.9: Working of Master-Slave JK Flip-flop

8.5. D flip-flop

D flip-flop is also known as data flip-flop or transparent flip-flop. In this type of flip-flop, when the clock pulse is active, the output follows its data input. The symbol of the D flip-flop is shown in the [figure](#)

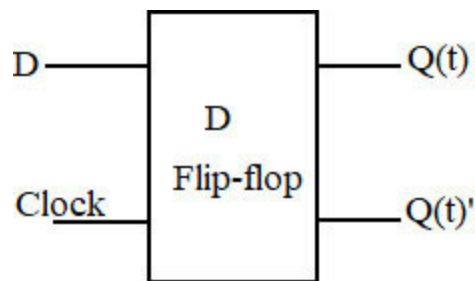


Figure 8.10: Symbol of D flip-flop

The D flip-flop can be constructed using NAND or NOR gate because both gates are having the capability of a universal gate. The [figure 8.11](#) shows D flip-flop using NAND gates.

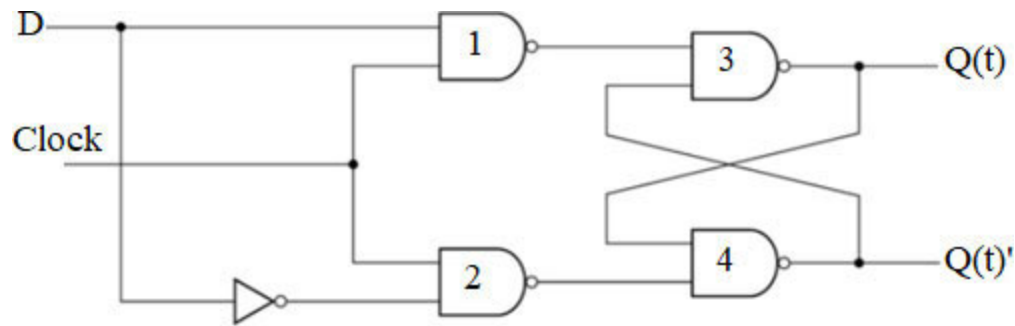


Figure 8.11: D flip-flop using NAND gates

Case (I): $D=0$

When $D=0$ and $\text{clock}=1$, the output of gate1 will be 1, and gate2 will be 0. Assume $Q(t)=0$ and $Q(t)'=1$, the value of $Q(t)'$ is fed to gate3 along with the output of gate1, which implies $Q(t)=0$. So, if $D=0$, then the calculated $Q(t)$ is also 0. It means that the same data is transferred.

Case (II): $D=1$

When $D=1$ and $\text{clock}=1$, the output of gate1 will be 0, and gate2 will be 1. Assume $Q(t)=0$ and $Q(t)'=1$, the value of $Q(t)'$ is fed to gate4 along with the output of gate2, which implies $Q(t)=1$. So, if $D=1$, then the calculated $Q(t)$ is also 1. It means that the same data is transferred. The output does not exist if the clock pulse is absent. The truth table of the D flip-flop is shown in [table](#)

Table 8.10: The truth table of D flip-flop using NAND gate

8.5.1. Characteristics table of D flip-flop

A characteristics function or equation of flip-flop is used to explore the value of the next state in terms of current state and output. It defines the logical property of flip-flops. [Table 8.11](#) shows the characteristics table of D flip-flop.

flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.

Table 8.11: Characteristics table of D flip-flop

The characteristics equation can be explored using k-map, as in the [figure](#)

		D	
		0	1
Q(t)	0		1
	1		1

Figure 8.12: K-map of D flip-flop characteristics table

$$Q(t+1)=D$$

8.5.2. Excitation table of D flip-flop

An excitation table of flip-flop shows the necessary input conditions for every possible transition of output. Based on the present and next state values, the input conditions are framed. The symbol 'X' signifies don't care condition, which means that it does not matter whether the input is 0 or 1. [Table 8.12](#) shows the excitation properties of D flip-flop.

flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.

Table 8.12: Excitation table of D flip-flop

The excitation table is deduced from the characteristics table. The characteristics table is looked for the values when $Q(t)=0$ and $Q(t+1) =0$, the corresponding values for this condition is $D=0$. Similarly, other values of the excitation table are filled.

Verilog program

```
module dff(q,clk,rst,d);  
input clk,rst,d;  
output reg q=0;  
always@(posedge clk or posedge rst)  
begin  
if(rst)  
q=0;  
else  
q=d;  
  
end  
endmodule
```

8.6. T flip-flop

T flip-flops are also known as toggle flip-flops. It is the modified form of JK flip-flop. It is designed by connecting J and K terminals together. The symbol of T flip-flop is shown in the [figure](#)

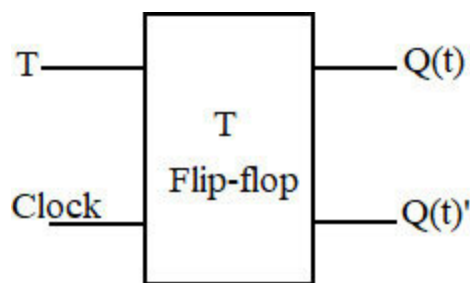


Figure 8.13: Symbol of T flip-flop

The T flip-flop can be constructed using NAND or NOR gate because both gates are having the capability of a universal gate. The [figure 8.14](#) shows T flip-flop using a NAND gate.

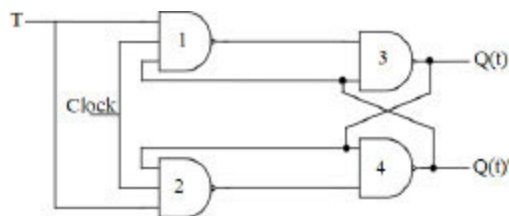


Figure 8.14: T flip-flop using NAND gates

Case (I): $T=0$

When $T=0$ and $\text{Clock}=1$, assume $Q(t)=0$ and $Q(t)'=1$. The output of gate1 and gate 2 will be at level 1. The value of $Q(t)'$ is fed to gate3 along with the output of gate1, it implies $Q(t)=0$, hence no change.

Case (II): $T=1$

When $T=1$ and $\text{Clock}=1$, assume $Q(t)=0$ and $Q(t)'=1$. The output of gate1 will be 0. The value of $Q(t)'$ is fed to gate3 along with the output of gate1; it implies $Q(t)=1$, data is toggled. Similarly, the other case. Toggle means output complements the present state. [Table 8.13](#) shows the input-output relationship of T flip-flop.

flip-flop.
flip-flop.
flip-flop. flip-flop.
flip-flop.
flip-flop.

Table 8.13: The truth table of T flip-flop using NAND gate

8.6.1. Characteristics table of T flip-flop

A characteristics function or equation of flip-flop is used to explore the value of the next state in terms of current state and output. It defines the logical property of flip-flops. [Table 8.14](#) shows characteristics properties of T flip-flop.

flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.

Table 8.14: Characteristics table of T flip-flop

The characteristics equation can be explored using k-map, as in the [figure](#)

Q(t) \ T	0	1
0		1
1	1	

Figure 8.15: K-map of D flip-flop characteristics table

$$Q(t+1) = T \cdot \overline{Q(t)} + \bar{T} \cdot Q(t)$$

$$Q(t+1) = T \oplus Q(t)$$

8.6.2. Excitation table of T flip-flop

An excitation table of flip-flop shows the necessary input conditions for every possible transition of output. Based on the present and next state values, the input conditions are framed. The symbol 'X' signifies don't care condition, which means that it does not matter whether the input is 0 or 1. [Table 8.15](#) shows the excitation properties of T flip-flop.

flip-flop.
flip-flop.
flip-flop.
flip-flop.
flip-flop.

Table 8.15: Excitation table of T flip-flop

The excitation table is deduced from the characteristics table. The characteristics table is looked for the values when $Q(t)=0$ and $Q(t+1) =0$, the corresponding values for this condition is

T=0. Similarly, other values of the excitation table are filled. For T=0, no change, and T=1, the toggle is applicable.

Verilog program

```
module tff(q,clk,rst,t);
input clk,rst,t;
output reg q=0;
always@(posedge clk or posedge rst)
begin
if(rst)
q=0;
else if(!t)

q=q;
else
q=~q;
end
endmodule
```

Conclusion

In this chapter, we studied about various types of flip-flop and its implementation using Verilog HDL.

In the next chapter, we will study the significance, design, and implementation of shift register using Verilog HDL.

Questions

Differentiate between flip-flop and latch

Differentiate between asynchronous and synchronous reset.

How the flip-flop can store the data.

The JK flip-flop acts as a toggle flip-flop. Justify it.

Convert SR flip-flop into T flip-flop.

CHAPTER 9

Shift Registers Implementation Using Verilog HDL

9.1. Introduction to shift registers

Flip-flops are used to store 1-bit data. For storing a word, a group of flip-flops can be used. A register is a group of flip-flops. Shift registers are sequential logic circuits that are capable of storing and shifting the data. An n-bit register has a group of n flip-flops.

Structure

In this chapter, we will discuss the following topics:

Significance and classification of shift registers

Design N-bit shift registers using flip-flops.

Implementation of shift registers using Verilog HDL.

Objectives

After studying this unit, you should be able to:

Classification and understand the working of shift registers

Design of various N-bit shift registers such as SIPO, SISO, PISO, and PIPO using flip-flop.

Implementation of shift registers using Verilog HDL.

9.1.1 Classification of shift registers

Shift registers are used to store and transfer data. They are a combination of flip-flops. A group of flip-flops connected in a chain so that the output of one flip-flop becomes the input of another flip-flop. All the flip-flops are driven by a standard clock pulse. The basic types of shift registers are shown in [figure](#)

Serial-In-Serial-Out (SISO) register

Serial-In-Parallel-Out (SIPO) register

Parallel-In-Serial-Out (PISO) register

Parallel-In-Parallel-Out (PIPO) register

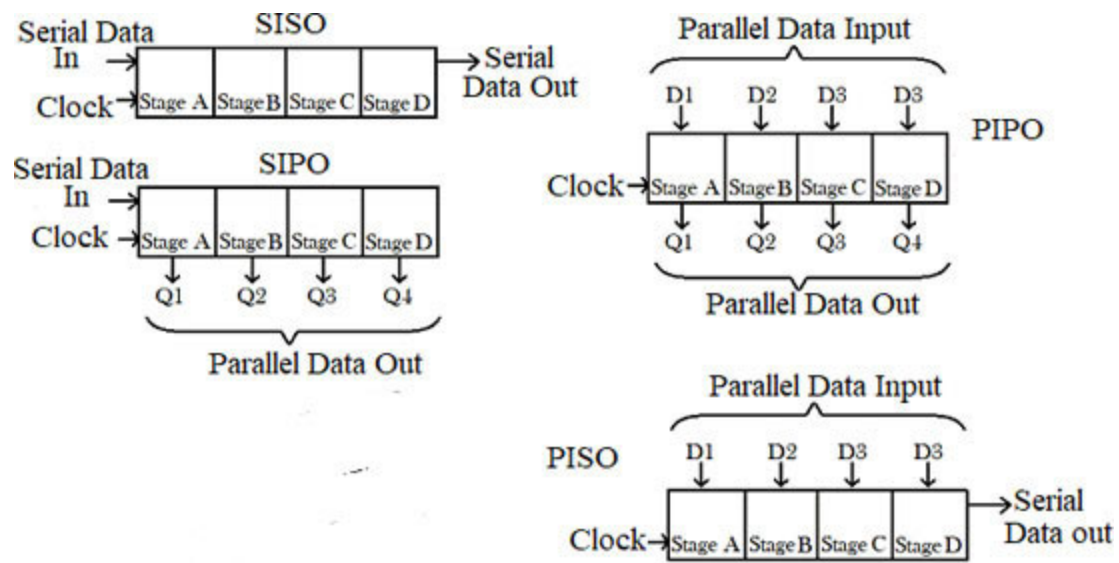


Figure 9.1: Types of shift registers

9.2. Serial-In-Serial-Out (SISO) shift register

A SISO register allows each flip-flop to pass the information from one flip-flop to its adjacent flip-flop serially, i.e., one bit by one bit. The movement of data from one end of a shift register to the other at a rate of one bit per clock pulse.

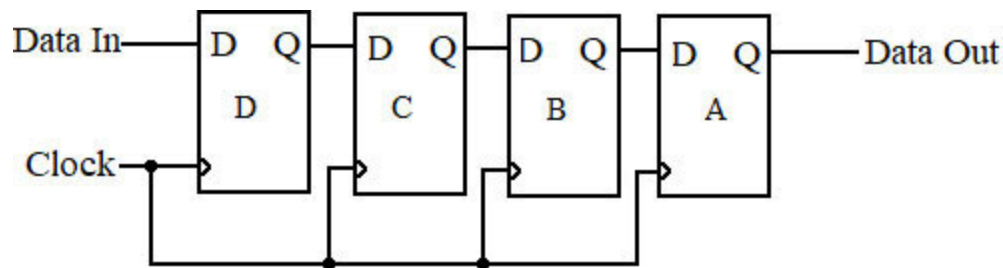


Figure 9.2: 4-Bit shift register (SISO)

Figure 9.2 represents a 4-bit serial-in-serial-out shift register, which consists of four D flip-flops. When the clock pulse is active, data from D flip-flop is shifted to its output Q. using clear input, all the flip-flops can be vacated initially. Let us take the case of data 1011. On the application of the first clock pulse, '1' is shifted to the output of first flip-flop, i.e., input of the second flip-flop. After completion of all the four clock pulses, '1' will reach the output of flip-flop4. In this way, the four-bit number is stored in a register. For extracting the data

out from the shift register, four more clock pulses will be needed.

The function of the SISO register is shown in [figure](#)

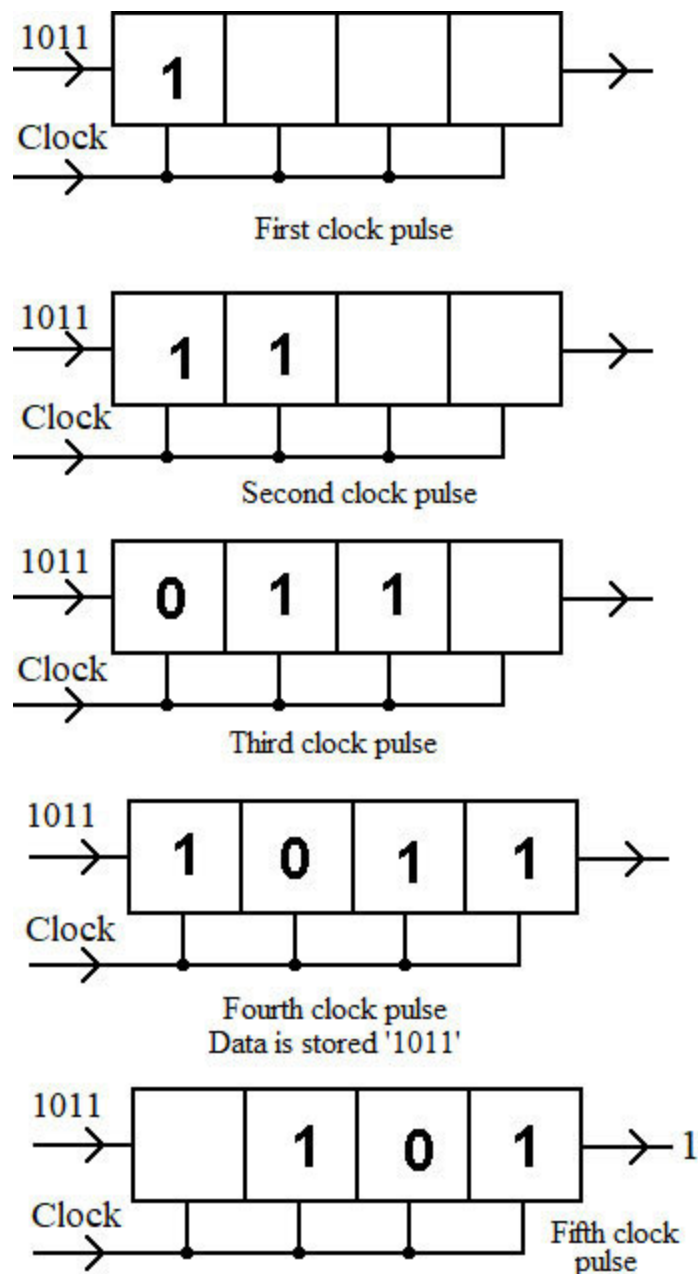


Figure 9.3: The functioning of the SISO register

So, for 4-bit data, storage requires four clock pulses, and data extracting needs four clock pulses.

Verilog program

```
module dff(d,clk,reset, q);
input d,clk,reset;
output reg q;
always @(posedge clk or posedge reset)
begin
if (reset)
q=0;
else
q=d;
end
endmodule
```

```
module SISO(d,clk,rst,q);
input d,clk,rst;
output q;
wire wo,w1,w2;
dff do(d,clk,reset,wo);
```

```
dff d1(w0,clk,reset,w1);  
dff d2(w1,clk,reset,w2);  
dff d3(w2,clk,reset,q);  
endmodule
```

9.3. Serial-In-Parallel-Out (SIPO) shift register

In the SIPO register, the data is fed to the flip-flops serially, i.e., one bit by one bit, and data is extracted parallelly. The number of flip-flops is equal to several bits to be stored. The block diagram and function of the SIPO register are shown in [figure 9.4](#) and [figure 9.5](#) respectively.

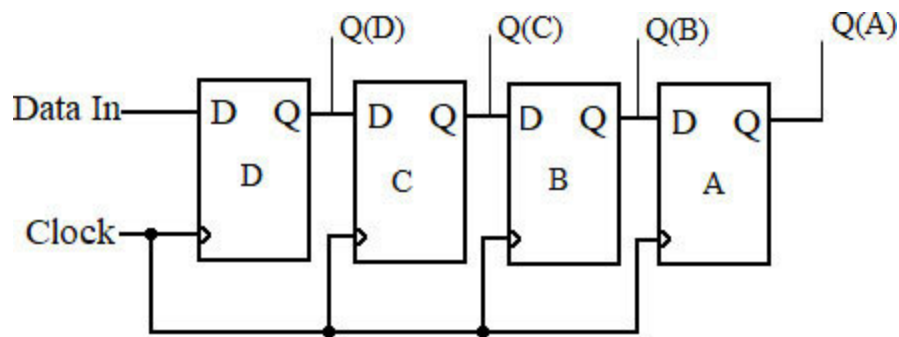


Figure 9.4: 4-bit shift register (SIPO)

Parallel shifting means the movement of data into all the flip-flops of a shift register at the same time, i.e., the output from all the flip-flops is extracted simultaneously.

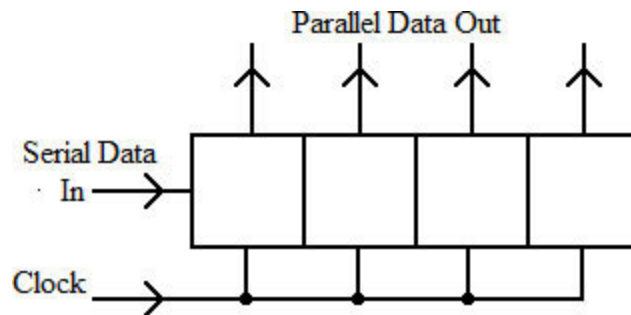


Figure 9.5: The functioning of the SIPO register

So, In the SIPO register, for 4-bit data, storage requires four clock pulses, and data extracting needs a single clock pulse. The 4-bit number will be extracted parallelly as a single word.

Verilog program

```

module dff(d,clk,reset, q);
input d,clk,reset;
output reg q;
always @(posedge clk or posedge reset)
begin
if (reset)
q=0;
else
q=d;
end
endmodule

```



```
module SIPO(d,clk,rst,q);  
input d,clk,rst;  
output [3:0]q;  
dff do(d,clk,reset,q[0]);  
dff d1(q[0],clk,reset,q[1]);  
dff d2(q[1],clk,reset,q[2]);  
dff d3(q[2],clk,reset,q[3]);  
endmodule
```

9.4. Parallel-In-Serial-Out (PISO) shift register

In PISO register, the data is fed to all the flip-flops simultaneously. While extracting the data from the flip-flops, it is a sequential process, i.e., one bit by one bit.

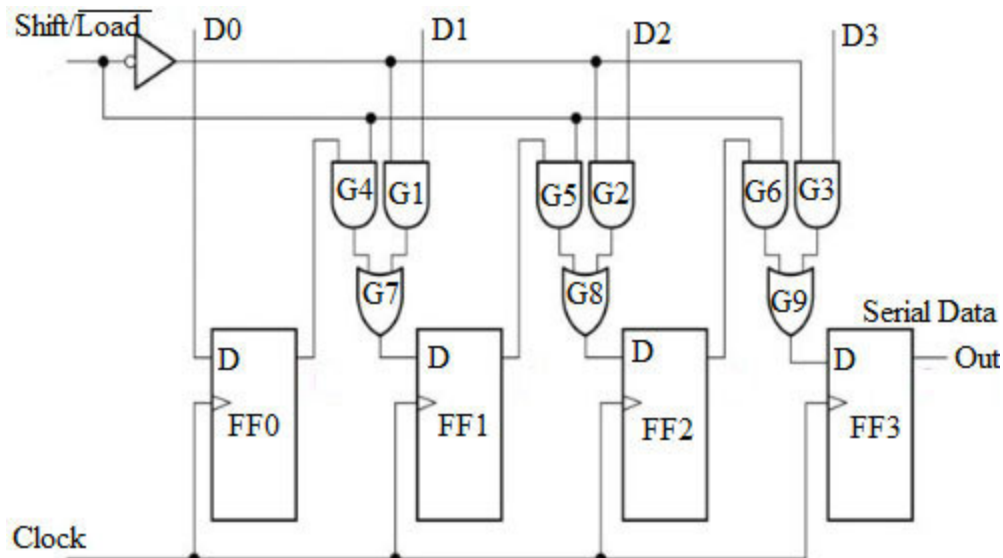


Figure 9.6: 4-bit shift register (PISO)

The block diagram and function of the PISO register are shown in [figure 9.6](#) and [figure 9.7](#) respectively. In a 4-bit PISO register, four flip-flops are used. Data D₀ D₁ D₂ D₃ is entered parallelly. is the control unit.

When is low, i.e., '0', then logic gates G₁, G₂, and G₃ are enabled, which allow input data to be applied to D input of its respective flip-flop. When the clock pulse is applied, the flip-flop with value D=1 will set, and D=0 will reset.

When is high, i.e., '1', then logic gates G₁, G₂, and G₃ are disabled, and logic gates G₄, G₅, and G₆ are enabled. The above-mentioned process allows the shifting operation.

The gates G₇, G₈ and G₉ at the D inputs of the flip-flops allow either parallel data entry or shift operation, depending upon the gates that are enabled by control parameter.

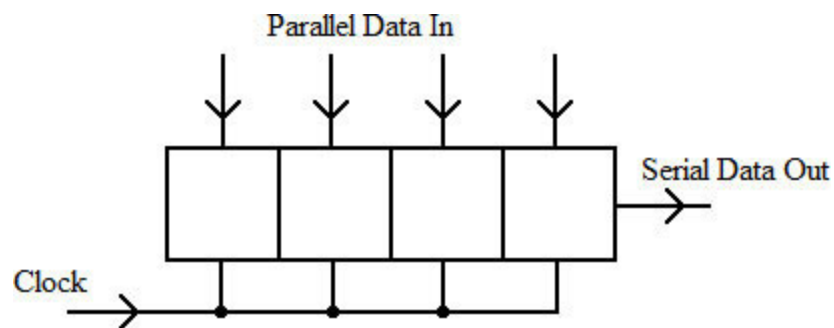


Figure 9.7: The functioning of the PISO register

So, In the PISO register, for 4-bit data, storage requires a single clock pulse for storing and four clock pulses for shifting operation.

Verilog program

```
module dff(d,clk,reset, q);
input d,clk,reset;
output reg q;
always @(posedge clk or posedge reset)
begin
if (reset)
q=0;
else
q=d;
end
endmodule
```

```
module SHIFT_LOADbar(a,b,S_Lbar,y);
input a,b,S_Lbar;
output y;
wire wo,w1,w2;
not n1(wo, S_Lbar);
and a1(w1,a, S_Lbar);

and a2(w2,b, wo);
or o1(y,w1,w2);
endmodule
```

```
module PISO(d,clk,rst,q,S_Lbar);
input clk,rst,S_Lbar;
```

```
input [3:0]d;
output q;
wire [5:0]w;
dff do(d[0],clk,reset,w[0]);
SHIFT_LOADbar so(w[0],d[1],S_Lbar,w[1]);
dff d1(w[1],clk,reset,w[2]);
SHIFT_LOADbar s1(w[2],d[2],S_Lbar,w[3]);
dff d2(w[3],clk,reset,w[4]);
SHIFT_LOADbar s2(w[4],d[3],S_Lbar,w[5]);
dff d3(w[5],clk,reset,q);
endmodule
```

9.5. Parallel-In-Parallel-Out (PIPO) shift register

PIPO register is a type of shift registers in which data are entering and data extracting are parallel processes. The PIPO can be pre-set to any value by directly loading a binary number into its internal flip-flops and also extract the output simultaneously. The block diagram and function of the PIPO register are shown in [figure 9.8](#) and [figure](#) respectively.

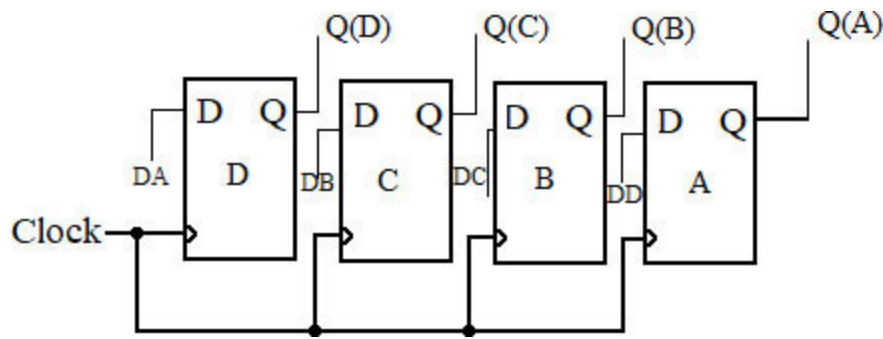


Figure 9.8: 4-bit shift register (PIPO)

Parallel shifting means the movement of data into all the flip-flops of a shift register at the same time, i.e., the output from all the flip-flops are extracted simultaneously.

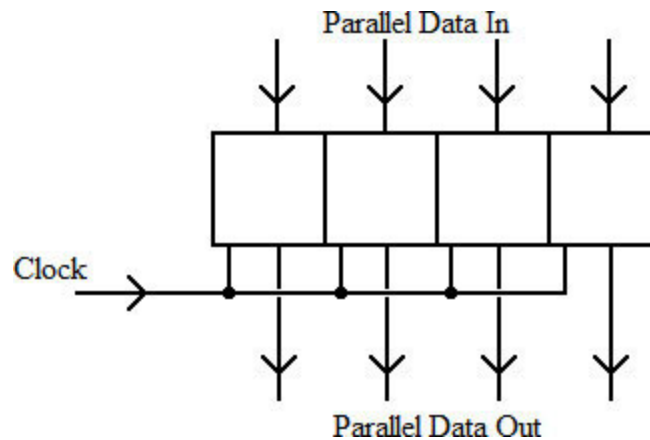


Figure 9.9: The functioning of the PIPO register

So, in a 4-bit PIPO shift register operation, one clock pulse is required for data entering, and one clock pulse is required for the extraction of data. In total, two clock pulses are consumed by the PIPO shift register for data store and shifting purpose.

Verilog program

```

module dff(d,clk,reset, q);
input d,clk,reset;
output reg q;
always @(posedge clk or posedge reset)
begin
if (reset)
q=0;
else
q=d;

```

```
end  
endmodule
```

```
module PIPO(d,clk,rst,q);  
input clk,rst;  
input [3:0]d;  
output [3:0]q;  
dff do(d[0],clk,reset,q[0]);  
dff d1(d[1],clk,reset,q[1]);  
dff d2(d[2],clk,reset,q[2]);  
dff d3(d[3],clk,reset,q[3]);  
endmodule
```


Conclusion

In this chapter, we studied various types of the shift register and its implementation using Verilog HDL.

In the next chapter, we will study the design and implementation of counters using Verilog HDL.

Questions

Explain the operation of the bi-directional shift register.

What is the difference between serial transfer and parallel transfer?

What is the function of the PISO shift register? How many clock pulses are needed for input and output data in PISO, if data is 10010?

Differentiate between SISO and PIPO with an example.

CHAPTER 10

Counter Implementation Using Verilog HDL

10.1. Introduction to Counters

A counter is a sequential circuit that is capable of counting the number of clock pulses arriving at its clock input. A counter is a set of flip-flops whose states change in response to pulses applied at the input of the counter. The flip-flops are interconnected, such that their combined state at any time is the binary equivalent of the total number of pulses that have occurred up to that time. One of the applications of the counter is frequency divider.

An n-bit counter consists of n flip-flops and can count from 0 to $2^n - 1$. It is also called the Mod- 2^n counter, where 2^n is the number of states.

Structure

In this chapter, we will discuss the following topics:

Significance and classification of counters

Understand the concept of synchronous and asynchronous counters

Design and implementation of counters using Verilog HDL.

Objectives

After studying this unit, you should be able to:

Working and design of asynchronous and synchronous counters.

Design of Mod-N counters using flip-flop.

Implementation of various counters using Verilog HDL.

10.1.1. Classification of counters

Counters are divided into two categories:

Asynchronous counter

Synchronous counter

Asynchronous counters are also known as ripple counter, serial counter, or non-synchronous counter. In this type of counter, each flip-flop is triggered by the previous flip-flop, and the first flip-flop is triggered by an external clock pulse. In the synchronous counter, all the flip-flops are triggered simultaneously by a standard clock pulse.

10.2. Asynchronous counters

In asynchronous counters, the number of flip-flops depends on the number of states. The number of output states of the counter is called Modules (MOD). If we have three flip-flops, the maximum number of states will be eight. We can name that counter, which has eight output states, as MOD-8 counter.

10.2.1. MOD-4/2-bit asynchronous counter/ripple counter

In the 2-bit asynchronous counter, two flip-flops are used. As the number of flip-flops is two, the possible states are $2^2=4$. Depending on the number of states, it is known as the MOD 4 counter.

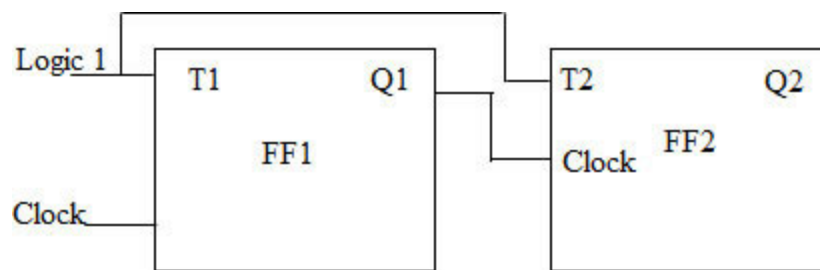


Figure 10.1: Asynchronous MOD-4 counter using T flip-flop

Figure 10.1 shows Asynchronous MOD-4 counter using T flip-flop. Initially, both flip-flops are cleared, i.e., $Q_2Q_1=00$.

As soon as the negative edge clock pulse is applied, FF1 will toggle and $Q_1=1$. This output Q_1 is connected to the second flip-flop as a clock. With the transition, it becomes a positive edge clock, so there will be no change in Q_2 . In such a manner, $Q_2Q_1=01$.

When the second clock pulse is applied, the FF1 will toggle, $Q_1=0$, which is connected to FF2. The change will act as a negative edge clock pulse, FF2 will toggle and $Q_2=1$. In such a manner, $Q_2Q_1=10$.

When the third clock pulse is applied, FF1 will toggle and $Q_1=0$. This output Q_1 is connected to the second flip-flop as a clock. With the transition, it becomes a positive edge clock, so there will be no change in Q_2 . In such a manner, $Q_2Q_1=11$

When the fourth clock pulse is applied, FF1 toggles again and $Q_1=1$. The change will act as a negative edge clock pulse; FF2 will toggle and $Q_2=0$. In such a manner, $Q_2Q_1=00$.

[illegible]

Table 10.1: The truth table of MOD-4 Asynchronous counter

The input-output relationship of the MOD-4 asynchronous counter is shown in [table](#)

10.2.2. Advantages & disadvantages of asynchronous counters

Advantages

Simple design

Number of components are less

Working is understandable easily

Disadvantages

As flip-flops are not clocked simultaneously, so the speed is slow.

For the truncated sequence, these counters are forced to recycle before going through all of its normal states.

The propagation delay is high. It limits the rate at which counter can be clocked.

10.2.3. Verilog_program of the 2-bit asynchronous counter using the positive edge-triggered T flip-flop

```
module tff(q,clk,rst,t);
input clk,rst,t;
output reg q=0;
always@(posedge clk or posedge rst)
begin
if(rst)
q=0;
else if(!t)
q=q;
else
q=~q;
end
endmodule
```

```
module counterASYN(clk,rst,q);
input clk,rst;
output [1:0]q;
tff to(q[0],clk,rst,1'b1);
tff t1(q[1],~ q[0],rst,1'b1);
endmodule
```

10.3. Synchronous counter

If the clock pulse is applied to all flip-flops simultaneously, then it is known as a synchronous counter, i.e., when the same clock pulse triggers all the flip-flops, it is known as a synchronous counter. In synchronous counters, the number of flip-flops depends on the number of states. The number of output states of the counter is called Modules (MOD). If we have two flip-flops, the maximum number of states will be four. We can name that counter, which has four output states, as MOD-4 counter.

10.3.1. Mod-4/2-bit synchronous counter using JK flip-flop

In the 2-bit synchronous counter, two flip-flops are used. As the number of flip-flops is two, the possible states are $2^2=4$. Depending on the number of states, it is known as the MOD 4 counter.

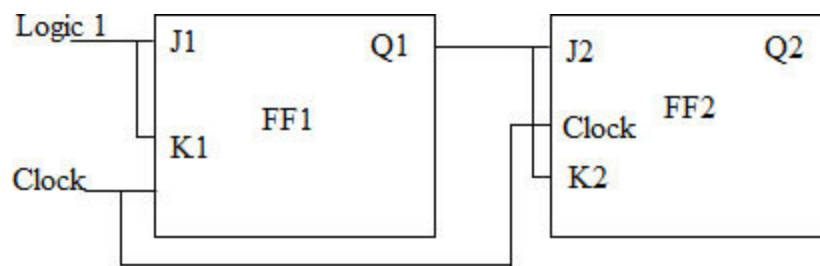


Figure 10.2: Synchronous MOD-4 counter using JK flip-flop

Figure 10.2 represents synchronous MOD-4/2-bit synchronous counter using JK flip-flop. The input J and K are connected and tied at logic 1. The same clock pulse is given to FF1 and FF2.

Initially, both flip-flops are cleared, i.e., $Q_2Q_1=00$.

As soon as negative edge clock pulse is applied, FF1 will toggle, because $J_1=K_1=1$, whereas Q2 will remain unchanged because of $J_2=K_2=0$ and $Q_1=1$, at the end of the pulse. So, after first clock pulse $Q_2Q_1=01$

When the second clock pulse is applied, both FF1 and FF2 will toggle. Thus, after the second clock pulse, $Q_2Q_1=10$

When the third clock pulse is applied, FF1 will toggle and $Q_1=1$. There will be no change in Q_2 . In such a manner, $Q_2Q_1=11$

When the fourth clock pulse is applied, FF1 and FF2 will toggle again as $J_1=K_1=J_2=K_2=1$. This results in $Q_2Q_1=00$. So, the counter is recycled back to its original state.

[illegible]

Table 10.2: The truth table of MOD-4 Synchronous counter

Table 10.2 shows the input-output relationship of the MOD-4 synchronous counter.

10.3.2. 3-bit up/down counter using T flip-flop

An up/down counter is also known as a bidirectional counter. As both operations up and down have been performed in a single circuit, so its operation is controlled by When this signal is low, decremented/down operation will be performed, and when this signal is at a high level, up/incremented operation will be performed.

State diagram

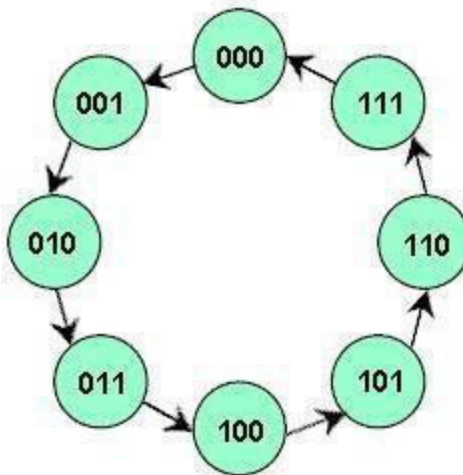


Figure 10.3: State diagram of the 3-bit counter

State table

The state diagram and state table of the 3-bit up/down counter shown in [figure 10.3](#) and [table 10.3](#) respectively.

respectively. respectively.

respectively.

respectively.

respectively.

respectively.

respectively.

respectively.

respectively.

respectively.

respectively.

respectively.

respectively.

respectively.

respectively.

respectively.

respectively.
respectively.
respectively.

Table 10.3: State table 3-bit Up/down counter

The logical expressions can be found using K-map.

$U_D Q_C \backslash Q_B Q_A$	00	01	11	10
00	1			
01	1			
11			1	
10			1	

$$TC = \overline{U_D} \cdot \overline{Q_B} \cdot \overline{Q_A} + U_D \cdot Q_B \cdot Q_A$$

$U_D Q_C \backslash Q_B Q_A$	00	01	11	10
00	1			1
01	1			1
11		1	1	
10		1	1	

$$TB = \overline{U_D} \cdot \overline{Q_A} + U_D \cdot Q_A$$

$U_D Q_C \backslash Q_B Q_A$	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

$$TC = 1$$

Figure 10.4: K-map implementation of 3-bit up/down counter

After extracting the logical expression, the 3-bit up/down synchronous counter is designed using T flip-flop as in [figure](#)

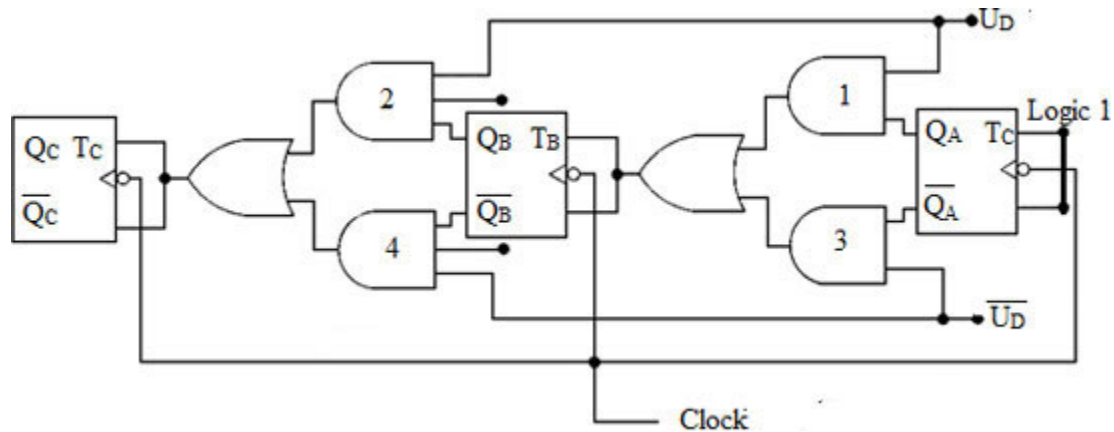


Figure 10.5: 3-bit synchronous up/down counter using T flip-flop

[Figure 10.5](#) shows the design of a 3-bit up/down counter, which performs increment as well as decrement operation, using the same logic circuit.

10.3.3. Verilog program of 2-bit synchronous counter using positive edge triggered D flip-flop

Positive edge triggered D flip-flop

```
module dff1(d,clk,rst,Q);
input d,clk,rst;
output reg Q;
always@(negedge clk or posedge rst)
begin
if (rst)
Q=0;
else
Q=d;
end
endmodule
```

Synchronous counter

```
module counterSYN(clk,rst,Q);
input clk,rst;
output reg[1:0]Q;
wire [1:0]w;
not n1(w[0],Q[0]);
```

```
xor x1(w[1],Q[o],Q[1]);  
dff1 d1(w[o],clk,rst,Q[o]);  
dff1 d2(w[1],clk,rst,Q[1]);  
endmodule
```

10.3.4. Verilog program of 4-bit up/down counter in behavioural modelling

```
module up_down(clk,rst,U_Dbar,q);
input clk,rst,U_Dbar;
output reg [3:0] q=4'b0000;
always@(posedge clk or posedge rst)
begin
if(rst)
q=4'b0000;
else if(U_Dbar)
q=q+1;
else if(!U_Dbar)
q=q-1;
else
q=q;
end
endmodule
```


Conclusion

In this chapter, we studied design and classification of various types of counters and its implementation using Verilog HDL.

In the next chapter, we will study about shift registers counters and its implementation using Verilog HDL.

Questions

Difference between registers and counters.

Differentiate between synchronous and asynchronous counters.

Design 4-bit Up counter using D flip-flop.

Design following counters using D flip-flop

Mod-7 counter

3-bit up/down counter

What is the difference between SISO and ring counter?

CHAPTER 11

Shift Register Counter Implementation Using Verilog HDL

11.1. Introduction to shift register counters

In the previous chapter, we have studied about counters and its implementation using Verilog HDL. Now, we will understand how shift register counters are different from counters. So, shift register counters are designed using SISO shift register with feedback from the output of the last flip-flop to the input of the first flip-flop. As they exhibit a specified sequence of states, therefore such devices are known as counters. In this chapter, the N-bit design of shift register counters using flip-flops has been shown.

Structure

In this chapter, we will discuss the following topics:

Structure and classification of shift register counters.

Design and implementation of ring counter and Johnson counter using Verilog HDL.

Implementation of shift register counters using Task and Functions.

Objectives

After studying this unit, you should be able to:

Understand the difference between counters and shift register counters.

Classification and implementation of shift register counters using Verilog HDL.

Design N-bit ring counter and Johnson counter using flip-flop.

11.1.1 Types of shift register counters

Shift register are also used as counters. There are two types of counters based on the type of output from right most D flip-flop is connected to the serial input. These are Ring counter and Johnson Ring counter

Ring counter

Johnson counter

11.1.1.1 Ring counter

In the ring counter, several flip-flops depend on the number of data bits. The arrangement of flip-flop array is in the form of a ring, so it is known as a ring counter. The output of the first flip-flop is connected to the input of the second flip-flop and so on. The output of the last flip-flop becomes the input of the first flip-flop by a feedback manner. [Figure 11.1](#) shows a 4-bit ring-counter using D flip-flop.

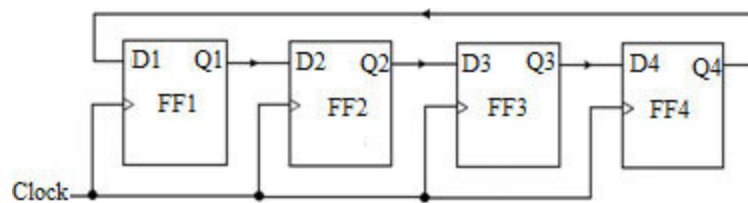


Figure 11.1: 4-bit ring counter

[Figure 11.1](#) shows a 4-bit ring counter, designed using D flip-flops. It is a circular shift register, where one flip-flop is set, and all other flip-flops are clear. One bit is shifted within the flip-flops to produce a sequence of timing signals. Let us take a case of 4-bit data 1100 to be shifted and circulated back with respect to clock signals. [Table 11.1](#) shows the sequence state table of the ring counter.

counter.
counter.
counter.
counter.
counter.
counter.

counter.
counter.
counter.

Table 11.1: Sequence table of the ring counter

The state diagram of the 4-bit ring counter is shown as in [*figure*](#)

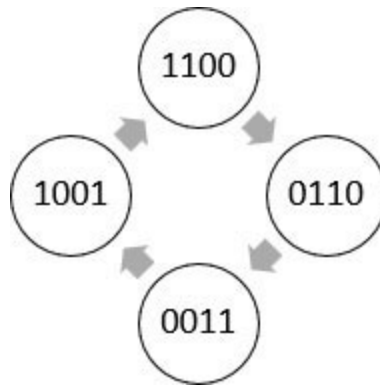


Figure 11.2: State diagram of the 4-bit ring counter

11.1.1.2. Johnson counter

The Johnson counter is also known as a twisted ring counter or switch-rail counter. The output of the first flip-flop is connected to the input of the second flip-flop and so on. The complemented output of the last flip-flop becomes the input of the first flip-flop by a feedback manner. Therefore, it is named as a twisted ring counter. The number of states can be doubled if the shift register is connected as a switch-trail ring counter.

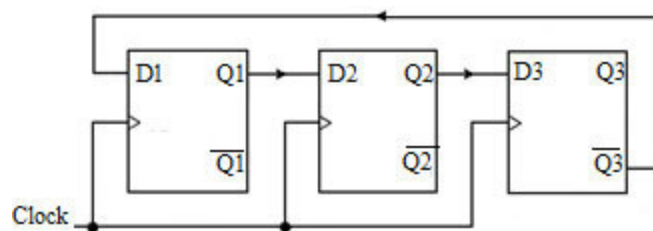


Figure 11.3: 3-bit Johnson counter

In [figure](#) 3-bit Johnson counter using D flip-flop is shown. The number of flip-flops is equal to several data bits. Let us reset all the flip-flops initially. The output $Q1$ is attached with $D2$, $Q2$ is connected with $D3$, and complemented output $Q3$

becomes the input for D₁. The sequence state table of the Johnson counter is shown in [table](#)

Table 11.2: Sequence table of Johnson counter

The state diagram of the 3-bit Johnson counter is shown in [figure](#)

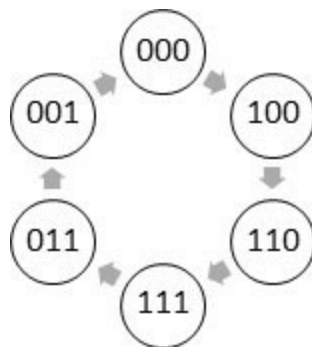


Figure 11.4: State diagram of 3-bit Johnson counter

11.2. Count the number of ones in a bit sequence using Verilog HDL

```
module countones(no_of_ones,number);
input [3:0]number;
output reg[2:0]no_of_ones;
integer i;
always @(*)
begin
no_of_ones=3'b000;
for (i=0;i<=3;i=i+1)
begin
if(number[i]==1)
begin
no_of_ones=no_of_ones+1'b1;
end
else
no_of_ones=no_of_ones;
end
end
endmodule
```

11.2.1. Implementation using TASK

```
module countonesTASK(no_of_ones,number);
input [3:0]number;
output reg[2:0]no_of_ones;
always @(*)
no_of_onesTASK (no_of_ones,number);
task no_of_onesTASK;
output [2:0]N_of_1;
input [3:0]no;
integer i;
begin
N_of_1=3'b000;
for (i=0;i<=3;i=i+1)
begin
if(no[i]==1)
begin
N_of_1=N_of_1+1'b1;
end
else
N_of_1=N_of_1;
end
end
endtask
```

endmodule

11.2.2. Implementation using FUNCTION

```
module countonesFUNC(no_of_ones,number);
input [3:0]number;
output reg[2:0]no_of_ones;
always @(*)
no_of_ones=no_of_onesFUNC(number);
function [2:0]no_of_onesFUNC;
input [3:0]no;
integer i;
begin
no_of_onesFUNC=3'b000;
for (i=0;i<=3;i=i+1)
begin
if(no[i]==1)
begin
no_of_onesFUNC=no_of_onesFUNC+1'b1;
end
else
no_of_onesFUNC=no_of_onesFUNC;
end
end
endfunction
endmodule
```


Conclusion

In this chapter, we studied the classification and design of shift register counters and their implementation using Verilog HDL.

In the next chapter, we will study advanced Verilog techniques. The user-defined primitives UDP will be discussed along with its Verilog HDL code.

Questions

How shift register counters and shift register are interconnected?

Differentiate between ring counter and Johnson counter?

Design 2-bit ring counter using D-flip-flop?

Design 3-bit Johnson counter. How it is different from the ring counter?

Design 3-bit ring counter using T flip-flop?

CHAPTER 12

Advanced Modelling Techniques

12.1. Introduction to UDP

Users can extend the set of Verilog predefined gate primitives by designing and specifying new primitive elements called User-Defined Primitives (UDPs). Instances of these new, self-contained, UDPs can then be used in the same manner as the gate primitives to represent the circuit being modeled. Employing UDPs can reduce the amount of memory needed for modeling and improve simulation performance.

Structure

In this chapter, we will discuss the following topics:

Classification of UDPs

Design of UDPs using Verilog HDL

Objectives

After studying this unit, you should be able to:

Understand the concept and syntax of combinational and sequential UDPs.

Design of combinational and sequential logic circuits using UDPs in Verilog HDL.

12.1.1. Types of UDP

UDPs are classified into two types:

Combinational UDP

Sequential UDP

Combinational UDP is defined where the output is solely determined by a logical combination of the inputs. The maximum number of inputs to a Combinational UDP is ten. Sequential UDP takes the value of its inputs and the current value of its output to determine the next value of its output. The value of the output is also the internal state of the UDP. Sequential UDP provides an easy and efficient way to model sequential circuits such as flip-flops and latches. Sequential UDP allows the mixing of the level-sensitive and edge-sensitive constructs in the same description. The maximum number of inputs to a Sequential UDP is limited to nine because the internal state counts as an input.

Each UDP has exactly one output, which can be in one of three states: 0, 1 or X. Tri-state value Z is not supported.

12.2. Syntax of UDP

The formal syntax of the UDP definition is as follows:

```
primitive (  
); // UDP name & terminal list  
output; // Terminals declarations  
input ;  
reg ; // optional, only for seq UDP  
initial = ;// only for seq UDP  
table // UDP state  
tableentries>  
endtable  
endprimitive // End of UDP definition
```

12.3. Rules of UDP

The following are the rules of UDP:

UDP can take only scalar input terminals (1 bit)

UDP can have only one scalar output. The output terminal must always appear first in the terminal list.

The state in a sequential UDP can be initialized with an initial statement.

State table entries can contain values of 0, 1 or X.Z values passed to a UDP are treated as X values.

UDPs are defined at the same level as modules.

UDPs are instantiated exactly like gate primitives.

UDPs do not support inout ports.

12.4. Verilog HDL program using UDP

The primitives that can be defined by the user is known as UDPs. By using UDPs, it is easier and efficient to design sequential and combinational logic circuits in Verilog HDL.

12.4.1. Verilog code for 4x1 multiplexer using UDP

Multiplexer is a combinational logic circuit that is designed to choose one input out of various inputs. Here, 'various' refers to 2^n , n is number of data inputs. The function of multiplexer is controlled by select line or control signal. The shortened form of multiplexer is MUX or MPX. The multiplexer is also known as data selector, as it 'selects' input data, to be produced on output terminal. In 4x1 multiplexer, there is four inputs with a single output.

```

primitive mux_4x1(muxed_out, sel_1, sel_0, data1, data2, data3, data4);
    output muxed_out;
    input sel_1, sel_0;
    input data1, data2, data3, data4;
    table
    // sel_1 sel_0 data1 data2 data3 data4 : muxed_out;
        0      0      1      ?      ?      ?      :      1;
        0      0      0      ?      ?      ?      :      0;
        0      1      ?      1      ?      ?      :      1;
        0      1      ?      0      ?      ?      :      0;
        1      0      ?      ?      1      ?      :      1;
        1      0      ?      ?      0      ?      :      0;
        1      1      ?      ?      ?      1      :      1;
        1      1      ?      ?      ?      0      :      0;
        ?      ?      1      1      1      1      :      1;
        ?      ?      0      0      0      0      :      0;
    endtable
endprimitive

```

12.4.2. Verilog code for JK flip-flop using UDP

The problem with SR flip-flop is its undetermined state at values $S=1$ and $R=1$. JK flip-flop is refined flipflop, in which an undetermined state of SR flip-flop is defined. The symbol of JK flip-flop is shown in [figure](#) where inputs are J and K with corresponding outputs $Q(t)$ and $Q(t)'$. The JK flip-flop is a combination of SR flip-flop having feedback. The letters J and K are chosen by inventor Jack Kirby.

```
primitive jkff_udp (q,clk,j,k);
input clk,j,k;
output q;
reg q;
table
//   clk   j    k    :    q    :    q+
    r     0    0    :    ?    :    - ;
    r     0    1    :    ?    :    0 ;
    r     1    0    :    ?    :    1 ;
    r     1    1    :    0    :    1 ;
    r     1    1    :    1    :    0 ;
    f     ?    ?    :    ?    :    - ;
    ?     *    ?    :    ?    :    - ;
    ?     ?    *    :    ?    :    - ;
endtable
endprimitive
```


Conclusion

In this chapter, we studied about the use of UDPs in combinational and sequential logic circuits, during implementation using Verilog HDL.

In the next chapter, we will study about the switch level modeling and CMOS implementation of various circuits and functions.

Questions

Name any two combinational and sequential UDPs.

How UDPs are different than inbuilt primitives.

Name three in-built primitives of Verilog HDL.

Design 8x1 multiplexer using UDPs.

CHAPTER 13

Switch Level Modelling

13.1. Introduction to switch level modeling

The transistor-level modeling or switch-level modeling is hardware modeling using transistors or switches to implement digital designs. Therefore, the basic unit in switch-level modeling is transistors or switches. The complexity of the design increases as we go with switch-level modeling. Due to this, switch-level modeling is said to the lowest level of abstraction in Verilog HDL. The logic values in Verilog HDL can consider only four digital values, i.e., 0, 1, Z, and X logic values.

Structure

In this chapter, we will discuss the following topics:

Basic digital designs using PMOS, NMOS & CMOS switches.

HDL programming using switch-level primitives.

Objectives

After studying this unit, you should be able to:

Describe what the structural modelling is.

Learn to design digital circuits using PMOS, NMOS & CMOS switches.

Describe how to instantiate switch primitives.

13.2. Switch level primitives

Verilog provides various primitives to design digital circuits in the lowest level of abstraction.

13.2.1.MOS switches

The P-channel MOSFET & N-channel MOSFET can be used in Verilog HDL using the 'nmos' and 'pmos' primitives. The symbols for NMOS and PMOS switches are shown in [figure](#) The d, s & g terminals in [figure 13.1](#) represent the drain, source & gate, respectively.

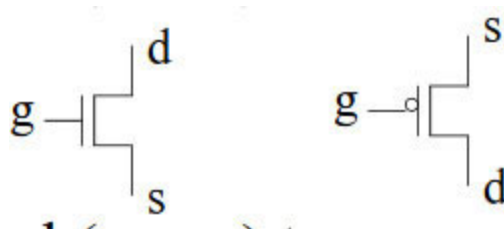


Figure 13.1: NMOS and PMOS Switches

In Verilog HDL, the syntax of NMOS and PMOS primitive are as below:

```
nmos x1(d, s, g); // Here 'd' act as the output, 's' act as  
the input & 'g' act as the control  
pmos x2(d, s, g); // Here 'd' act as the output, 's' act as  
the input & 'g' act as the control
```

Here instance name is not essential. Therefore, the simulator produces the result even without the instance name. However, it will issue a warning if the switch level primitives are instantiated without the instance name.

```
nmos (d, s, g); // Acceptable without instance name  
pmos (d, s, g); // Acceptable without instance name
```


13.2.2. CMOS switches

The Transmission Gate (TG) or CMOS switches (parallel connection of a PMOS & an NMOS switch, as shown in [figure](#)) can be used in Verilog HDL using the `cmos` primitive. The `out`, `data`, `pcontrol` & `ncontrol` represents the drain, source & gate, respectively.

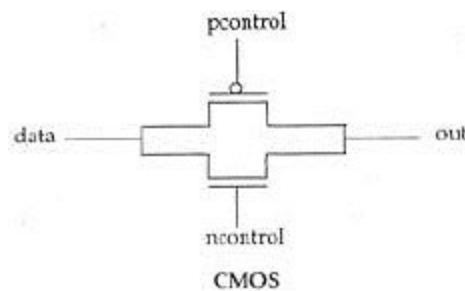


Figure 13.2: CMOS switch

A CMOS switch is instantiated as shown in below:

```
cmos cm1(d, s, ng, pg); // Here 'd' act as the output, 's'  
act as the input, 'ng' act as the ncontrol  
// 'pg' act as the pcontrol
```

The ncontrol and pcontrol are complement to each other. As mentioned in the case of NMOS & PMOS switches, CMOS switch also doesn't require instance name compulsorily to produce the result.

```
cmos (d, s, ng, pg); //no instance name given
```

13.3. Switch level modelling using Verilog HDL

This section shows the Verilog HDL program for two input nand gate and logical expression.

13.3.1. Two-input NAND gate using PMOS/NMOS switches

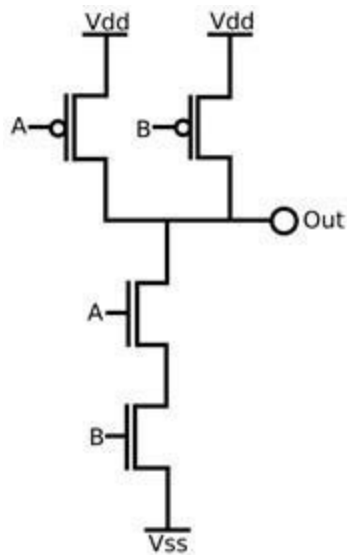


Figure 13.3: Switch level representation of two input NAND gate

```
module nand_switch (Out,A,B);  
input A,B;  
output Out;  
wire W;  
supply1 VDD;  
supply0 VSS;  
pmos pm1(Out,VDD,A)  
pmos pm2(Out,VDD,B);
```

```
nmos nm1(W,VSS,B);  
nmos nm2(Out,W,A);  
endmodule
```

13.3.2. Verilog HDL program in switch level modelling for implementing the logical expression, $Y = AB + C$

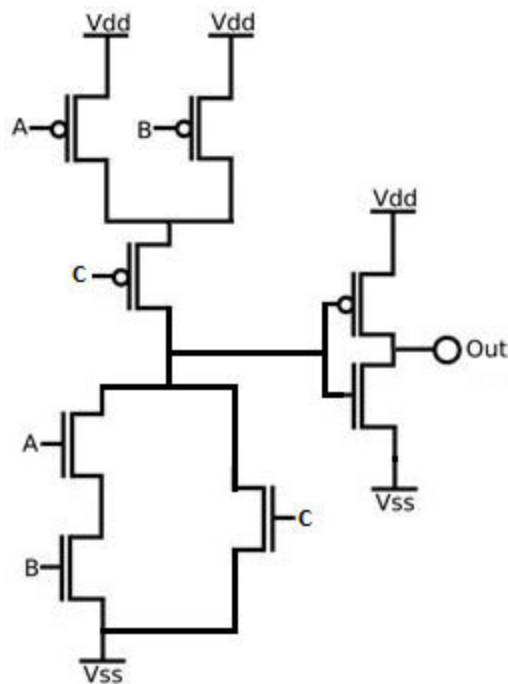


Figure 13.4: Switch level representation of logical expression

```
module design_switch (Out,A,B,C);  
input A,B,C;  
output Out;  
wire [2:0]w;  
supply1 VDD;  
supply0 VSS;
```

```
pmos p1(w[o],VDD,A)
```

```
pmos p2(w[o],VDD,B);
```

```
pmos p3(w[1],w[o],C);
```

```
nmos n1(w[2],Vss,B);
```

```
nmos n2(w[1],VSS,C);
```

```
nmos n3(w[1],w[2],A);
```

```
pmos p4(Out,VDD,w[1]);
```

```
nmos n4(Out,VSS,w[1]);
```

```
endmodule
```

13.3.3. Verilog HDL program in switch level modelling (using TG) for implementing the logical expression, $Y = AB + C$

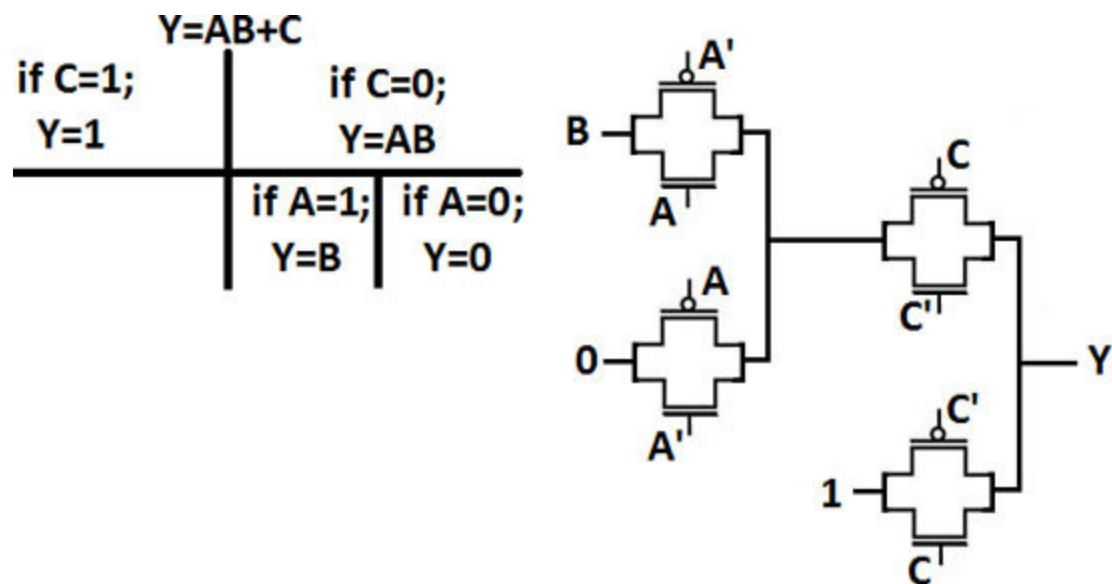


Figure 13.5: TG representation of logical expression

```

module design_TG (Out,A,B,C);
input A,B,C;
output Out;
wire W;
cmos c1(W,B,A,~A);
cmos c2(W,0,~A,A);
cmos c3(Y,W,~C,C);

```



```
cmos c4(Y,1,C,~C);  
endmodule
```

Conclusion

In this chapter, we studied the Verilog HDL programs using 'nmos,' 'pmos,' & 'cmos' primitives. The chapter also explains a few examples of NMOS-switches & CMOS switch.

In the next chapter, we will study the HDL programming using the Xilinx Design Constraint file on Nexys-4 Artix-7 FPGA Board.

Questions

Which level of abstraction is said to be the lowest level of abstraction in Verilog HDL?

List the differences between the MOS-switches & CMOS switch.

What is the relationship of 'ncontrol' & 'pcontrol' in CMOS switch?

Write a Verilog program to implement the function $Y = P'S + Q'R'$ using:

MOS-switches

CMOS switches

CHAPTER 14

FPGA Prototyping in Verilog HDL

14.1. Introduction to HDL programming

The structure and behavior of electronic circuits are described by HDL. It is a hardware description language that allows analysis and simulation of digital logic and circuits. An IC is created using synthesis, netlist generation, masking, and optimization. The HDL is an integral part of the EDA (Electronic Design Automation) tool for PLDs, microprocessors, and ASICs. The hardware description language was created to implement RTL (register transfer level) abstraction. The HDL is similar to language C. The first HDL was designed in the late 1960s. The HDL involves a testbench, which is designed to verify the correctness of the circuit. The HDL proves to be an excellent language for CPLD and FPGA. HDL includes a textual description consisting of operators, expressions, statements, inputs, and outputs. The widely used hardware description languages are Verilog, Very high-speed integrated circuit HDL (VHDL), System Verilog, Verilog C. With the advancement of technology, the language is improved at an accelerated rate. The latest iteration of Verilog, IEEE 1800-2005 System Verilog has improved features and architectures for design hierarchy and reuse.

Structure

In this chapter, we will discuss the following topics:

Specification and need for FPGA

Programming FPGA board using VIVADO

Objectives

After studying this unit, you should be able to:

Programming FPGA board

Digital design on Nexys-4 Artix-7 FPGA Board

14.2. Programming FPGA board

While programming FPGA board using VIVADO, the following files are required:

HDL source file

HDL top module

Xilinx Design Constraint (XDC) file

The HDL source file is a Verilog or VHDL file consists of the main module, and it's sub-modules. The top module is used to connect the HDL code's ports to the FPGA board's input/output pins. It acts as a bridge between the source code & the hardware. The XDC file has got significant importance for programming on high-end boards such as Nexys-4 Artix-7 FPGA boards. When programming on such boards using Xilinx's Vivado tool, you need to inform the software what physical pins on the FPGA that you plan on using or connecting to, with the HDL code that you wrote to describe the behavior of the FPGA. It is

a SCRIPT file which describes the physical pins on the FPGA.



Figure 14.1: Nexys-4 Artix-7 FPGA board

14.3. Example of digital design on Nexys-4 Artix-7 FPGA board

The half adder design has been implemented using the Nexys-4 Artix-7 FPGA board. It has three parts: HDL Source file generation, HDL Top file, and XDC file. The program code is described as,

14.3.1. Half adder

The half adder operates addition on two binary bits, augend, and addend. It produces output in the form of sum and carry.

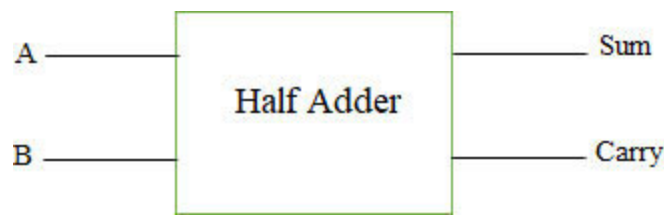


Figure 14.2: Block diagram of half adder

As per steps, the number of inputs and outputs are identified and labeled it. There are two inputs, A and B producing two outputs Sum and Carry. As the number of inputs is two, it suggests four possible combinations. The half adder performs binary addition. The block diagram of half adder is shown in [*figure*](#)

Table 14.1: The truth table of half adder

[Table 14.1](#) shows the input-output relationship of half adder. A minimized Boolean function can be obtained using a Karnaugh map. The two-variable k-map is used to explore the Boolean function.

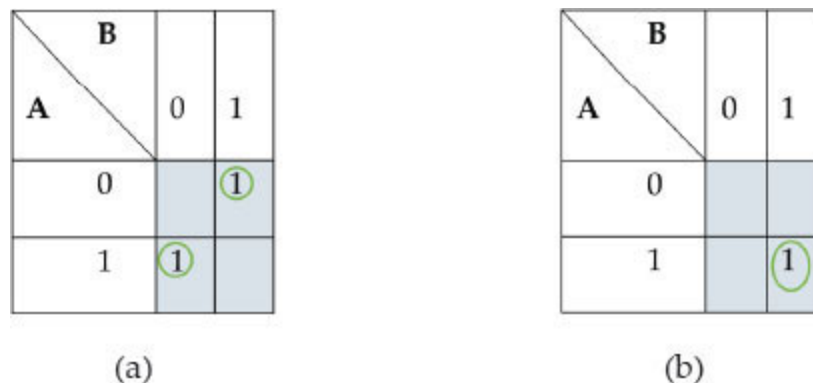


Figure 14.3: K-map for (a) Sum and (b) Carry

Using k-map, as in [figure](#) we get:

$$\text{Sum} = A\bar{B} + \bar{A}B \quad (14.1)$$

$$\text{Carry} = AB \quad (14.2)$$

The logical representation of Sum and carry using combination of EX-OR and AND gate, is shown in [figure](#)

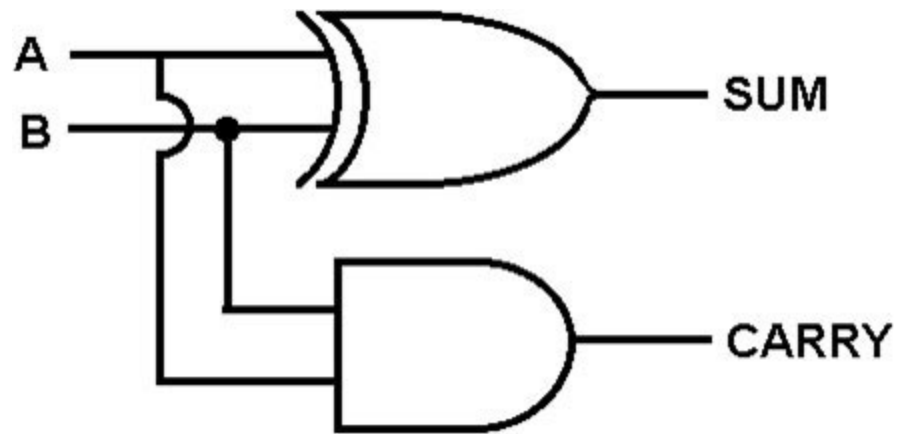


Figure 14.4: Half adder using logic gates

14.3.2. HDL source file

The HDL source file is a Verilog or VHDL file consists of the main module, and it's sub-modules. The HDL source file for half adder:

```
module HA(Sum,Carry,A,B);  
input A,B;  
output Sum,Carry;  
assign Sum=A^B;  
assign Carry=A&B;  
endmodule
```

14.3.3. HDL top module

The top module is used to connect the HDL code's ports to the FPGA board's input/output pins. It acts as a bridge between the source code & the hardware.

```
module Top(led,sw);  
input [1:0]sw;  
output [1:0]led;  
HA h1(led[0], led[1],sw[0],sw[1]);  
//vector part select & connection of the "LED" &  
"SWITCH" to the input/output port of "HA".  
endmodule
```


14.3.4. XDC file

The XDC file has got significant importance for programming on high-end boards such as Nexys-4 Artix-7 FPGA boards.

Switches

##Bank = 34, Pin name = IO_L21P_T3_DQS_34, Sch name = SWo

set_property PACKAGE_PIN U9 [get_ports {sw[o]}]

set_property IOSTANDARD LVCMOS33 [get_ports {sw[o]}]

##Bank = 34, Pin name = IO_25_34, Sch name = SW1

set_property PACKAGE_PIN U8 [get_ports {sw[1]}]

set_property IOSTANDARD LVCMOS33 [get_ports {sw[1]}]

LEDs

##Bank = 34, Pin name = IO_L24N_T3_34, Sch name = LEDo

set_property PACKAGE_PIN T8 [get_ports {led[o]}]

set_property IOSTANDARD LVCMOS33 [get_ports {led[o]}]

```
##Bank = 34, Pin name = IO_L21N_T3_DQS_34, Sch  
name = LED1  
set_property PACKAGE_PIN V9 [get_ports {led[1]}]  
set_property IOSTANDARD LVCMOS33 [get_ports {led[1]}]
```

Conclusion

In this chapter, we studied about the FPGA and understood the program coding and interconnection of FPGA using VIVADO.

Questions

Differentiate between configurable and reconfigurable computing.

Differentiate between FPGA and CPLD.

How ASIC is different from FPGA.

Give the specification chart of any two FPGA board